NAVAL POSTGRADUATE SCHOOL
Monterey, California

THESIS

THE DESIGN AND IMPLEMENTATION OF AN OBJECT-ORIENTED, PRODUCTION-RULE INTERPRETER

by

Heinz M. McArthur
December 1984

Thesis Advisor: Bruce J. MacLennan

Approved for public release; distribution is unlimited

SECURITY CLASSIFICATION OF THIS PAGE *(When Data Entered)*

REPORT DOCUMENTATION PAGE		READ INSTRUCTIONS BEFORE COMPLETING FORM
1. REPORT NUMBER	2. GOVT ACCESSION NO.	3. RECIPIENT'S CATALOG NUMBER
4. TITLE *(and Subtitle)* The Design and Implementation of an Object-Oriented, Production-Rule Interpreter		5. TYPE OF REPORT & PERIOD COVERED Master's Thesis December 1984 6. PERFORMING ORG. REPORT NUMBER
7. AUTHOR(s) Heinz M. McArthur		8. CONTRACT OR GRANT NUMBER(s)
9. PERFORMING ORGANIZATION NAME AND ADDRESS Naval Postgraduate School Monterey, California 93943		10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS
11. CONTROLLING OFFICE NAME AND ADDRESS Naval Postgraduate School Monterey, California 93943		12. REPORT DATE December 1984 13. NUMBER OF PAGES 211
14. MONITORING AGENCY NAME & ADDRESS*(if different from Controlling Office)*		15. SECURITY CLASS. *(of this report)* UNCLASSIFIED 15a. DECLASSIFICATION/DOWNGRADING SCHEDULE

16. DISTRIBUTION STATEMENT *(of this Report)*

Approved for public release; distribution is unlimited

17. DISTRIBUTION STATEMENT *(of the abstract entered in Block 20, if different from Report)*

18. SUPPLEMENTARY NOTES

19. KEY WORDS *(Continue on reverse side if necessary and identify by block number)*

object-oriented, relational, production-rule, rule-based, pattern-directed, pattern-matching

20. ABSTRACT *(Continue on reverse side if necessary and identify by block number)*

In this thesis we describe the design and implementation of two prototype interpreters for Omega, an object-oriented, production-rule programming language. The first implementation is a throw-away prototype written in LISP; the second implementation is a more complete version written in C. The Omega language features two major components: a set of production rules executed through pattern-directed invocation, and a relational database of values and objects. We develop a simple system of rule (Continued)

DD FORM 1 JAN 73 1473 EDITION OF 1 NOV 65 IS OBSOLETE

S/N 0102-LF-014-6601

ABSTRACT (Continued)

evaluation which relies on hashed indexing for rule selection and a list implementation of relations. The system's performance is evaluated in comparison with LISP and Prolog interpreters. We conclude with a discussion of our experience in developing example applications, and recommend extensions to the language based on this experience.

The Design and Implementation of an Object-Oriented, Production-Rule Interpreter

by

Heinz M. McArthur
Captain, United States Marine Corps
B.S., United States Naval Academy, 1977

Submitted in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE IN COMPUTER SCIENCE

from the

NAVAL POSTGRADUATE SCHOOL
December 1984

ABSTRACT

In this thesis we describe the design and implementation of two prototype interpreters for Omega, an object-oriented, production-rule programming language. The first implementation is a throw-away prototype written in LISP; the second implementation is a more complete version written in C. The Omega language features two major components: a set of production rules executed through pattern-directed invocation, and a relational database of values and objects. We develop a simple system of rule evaluation which relies on hashed indexing for rule selection and a list implementation of relations. The system's performance is evaluated in comparison with LISP and Prolog interpreters. We conclude with a discussion of our experience in developing example applications, and recommend extensions to the language based on this experience.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

I. INTRODUCTION

A. BACKGROUND

Two major issues in programming language design can be characterized as abstraction and architecture. In this context, abstraction refers to the ability of the language to capture the ideas of the programmer. It is a measure of expressiveness or semantic power. Architecture refers to those language characteristics, both organizational and syntactic, that affect practical usage. This includes ease of use for the programmer as well as the potential for efficient implementation. Thus an important goal for a programming language is to combine a powerful abstraction ability with an effective language architecture.

Conventional languages have suffered in both of these areas. These languages focus on the use of assignment statements for computation, and execution consists of the sequential flow of program control between assignment statements. John Backus described these "von Neumann" languages as excessively complex and weak, whose word-at-a-time conceptual basis has created an "intellectual bottleneck" [Ref. 1: p. 615]. These languages are oriented more towards the word-at-a-time stored program computer than towards the problem domains they attempt to satisfy. Thus they have poor abstraction ability. While simple, elegant imperative languages such as Pascal have enjoyed popularity, the need for increased power has resulted in complex languages such as Ada. Such languages have attained semantic power at the expense of architectural effectiveness.

Several alternatives to the von Neumann languages have been developed. Of interest in this thesis are applicative

languages, object-oriented languages, and rule-based inference languages.

B. APPLICATIVE LANGUAGES

Applicative languages use the application of a function to its arguments as the focus for computation. These languages are typified by pure LISP [Ref. 2], and by later functional languages such as FP [Ref. 1] and KRC [Ref. 3].

The strengths of the applicative languages are exemplified by arithmetic expressions. These strengths include clear interfaces between computational units, relative independence of evaluation order, and a semantic regularity that lends itself to simple verification and proof techniques.

Functionals, functions which receive functions as arguments and return functions as results, provide a mechanism for the combination of simple, primitive computational units into collections of arbitrary power and complexity.

The applicative languages achieve their predictability largely from the prohibition of side-effects during computation. This characteristic limits the problem domains to which applicative sclutions can be readily applied. Like the arithmetic expression, applicative languages cannot readily describe the notion of state. There is no explicit notion of time in an arithmetic expression, and applicative languages are correspondingly weak in maintaining temporal relationships. This characteristic limits the utility of applicative languages in inherently state-oriented applications. Such applications include operating system activities, data base management, and discrete simulation.

C. OBJECT-ORIENTED LANGUAGES

The object-oriented languages have developed from simulation languages such as Simula [Ref. 4], and are typified

by Smalltalk [Ref. 5]. Smalltalk partitions the programmer's model into collections of objects called classes. Objects have a state associated with them, and the methods (procedural information) of the class determine an object's computational behavior. Both data and procedural information are organized around the object.

The object-oriented approach allows certain important capabilities. Foremost, the concept of state is fully ingrained in the language. The simulation approach facilitates the modeling of real-world activities, with concurrency readily handled through the mechanism of communicating objects.

Associated with the class mechanism in Smalltalk is the concept of inheritance. When a new object is created, it obtains certain default state and behavioral characteristics from its class. In the functional languages, combinatorial power is obtained through subordinate function application and the use of functionals. In the object-oriented approach, combinatorial power is obtained through composition of new objects from existing ones, and through inheritance.

D. INFERENCE SYSTEMS AND LOGIC PROGRAMMING

Inference systems have developed through artificial intelligence efforts at cognitive modeling, knowledge representation, and theorem proving. Based on the early production system of Post [Ref. 6], these systems use rules, similar to logical implication, that provide the computational framework for a program. Rule-based organization has been described by Newell [Ref. 7], and an early language based on the concept was Hewitt's PLANNER [Ref. 8]. Numerous rule-based systems have since been developed, with notable examples being theorem provers such as AM [Ref. 9],

and expert systems such as MYCIN [Ref. 10], DENDRAL [Ref. 11], and PROSPECTOR [Ref. 12].

Prolog is a general-purpose language which uses rule-based theorem proving as the computational metaphor [Ref. 13]. Distinct from the applicative languages, Prolog uses pattern-matching instead of the procedure call to determine the applicability of rules and the resulting computations.

E. A COMBINED APPROACH

The preceding discussion highlights the following features offered by these languages:

- Function application provides a powerful, regular mechanism for stateless computation.
- An object-oriented approach provides an effective organization for data and procedures which is useful in representing temporal relationships and real-world objects.
- Rule-based pattern-matching systems have provided an alternative way for expressing complex knowledge representation.

The Omega language [Ref. 14] represents an approach that combines these features into a single language framework.

F. AN IMPLEMENTATION STUDY

The emphasis of this thesis is on the implementation of an interpreter for the Omega language. As an implementation study, the focus is on language architecture--those characteristics of the language that were conducive or that presented obstacles to efficient implementation. Of particular interest in this effort are the characteristics of

interpreter organization, data representation, and control strategies used in the implementation, and how these characteristics impact on the performance of the system.

This work is a prototyping effort. Because of the experimental nature of the language, various extensions and modifications were required on preliminary designs. To support this experimentation, two prototypes for the interpreter were written. The first was a throw-away prototype written in Franz LISP. The second was a more complete, incremental development written in C.

G. A SUBJECTIVE EVALUATION

As a secondary emphasis, some attention is given in this work to the evaluation of Cmega as a general-purpose programming language. While these observations are largely subjective, they provide some insight into the implementation problems associated with these early prototypes. Having prototype interpreters up and running has also provided an opportunity for experimentation with Omega programming that may prove useful in the early evaluation of features in the language.

II. AN INFORMAL DESCRIPTION OF THE LANGUAGE

A. GENERAL

This chapter provides a descriptive summary of the features of the Omega language. The descriptions are mainly by example, and serve to provide a feel for the language constructs, not detail.

The material in this chapter is based on the work of MacLennan. The philosophy, informal and formal semantics, and original syntax of the language are introduced in [Ref. 14]. Some syntactic and semantic differences exist between the original description of the language given in [Ref. 14] and the prototype implementations. Later chapters will deal with the rationale for these deviations. The implementation syntax is used in this chapter to maintain consistency with with the remainder of this thesis. A description of the implementation syntax is contained in Appendix A.

B. OBJECTS AND VALUES

The entities of the system are divided into values and objects. The values of the system include numerics (integers and reals), character strings, and lists. Lists are denoted by square brackets, such as:

```
["a", "b", [1, 2]].
```

Objects are referenced by name, and are subject to the following properties:

- Objects are unique.
- Objects may be shared.

- Objects are subject to change over time.
- Objects may be created and destroyed.

The distinction between objects and values is discussed in [Ref. 15]. Subsequent examples should help to illustrate the role of objects in Omega.

C. A RELATIONAL MODEL

The components of the language are organized according to a relational model. The model is consistent with relational terminology introduced by Codd [Ref. 16].

Consider an object that represents a queued process waiting for an operating system resource. For reference purposes, the object must be named, so call it **J**. Associated with the object are a priority, P, and a tape drive allocation requirement, T. The priority and resource requirements are values that must be associated with the job. These associations may be described by a **Priority** relation and a **TapeDrive** relation. This could be expressed as **Priority(J, P)**, **TapeDrives(J, T)**. In these expressions, the pairs **<J, P>** and **<J, T>** are called tuples.

A tuple is an ordered collection of objects and values. Note that, unlike relational database models, named attributes are not used to describe a tuple. Instead, the members of a tuple are described by relative position (order is important), by value, and by pattern-matching.

A relation is a set of tuples. Relations are described by name, and through pattern-matching. As a set, the tuples of a relation are (1) unique and (2) unordered.

Objects serve as representative place holders in relations. The state of an object is determined by the relations in which it participates, and by the attributes associated with the object in these relations.

Relations themselves are objects, although they are distinguished by having an intrinsic value: the collection of tuples instantiating the relation. As objects, relations may be members of tuples and participate in other relations.

D. PATTERN-DIRECTED PRODUCTION RULES

The relations organize the primitives of the system. Thus, at a given time, the state of the system is characterized by its relations and the entities bound through these relations. To complete the model, a mechanism must be used to describe state transitions. Pattern-directed production rules are used for this purpose.

A rule is a pair <a, c>, where a is termed an antecedent and c a consequent. An antecedent consists of Boolean conditions that pertain to the state of the system. The consequent consists of actions that will be executed if the conditions of the antecedent are true. Rules are written:

if <antecedent> -> <consequent>.

A condition may be one of several constructs. The most fundamental is the inquiry. An inquiry is a pattern-matched test described by the rule that is performed against the relations of the system. As an example, consider the job queue again. A rule may be desired that checks for jobs requesting 4 tape drives. This could be expressed as:

if TapeDrives(j, 4) -> . . .

where the consequent of the above rule is not shown. The expression TapeDrives(j, 4) is an inquiry, and may be read as "if there is a tuple <entity, 4> in the TapeDrives relation, then return true and bind the entity to the variable j." The symbol j in this example is an unbound logical variable, which is considered a wild card in an attempt to

match the inquiry against the tuples of the relation. Once a logical variable has become bound through an inquiry, this binding will remain in effect for that particular rule.

The following, more complex, inquiry relies on variable binding:

```
if TapeDrives(j, n), Priority(j, p) -> . . .
```

The comma is considered as a logical "and" between the two inquiries. Thus, the antecedent of this rule will be evaluated as true if tuples exist in the **TapeDrives** and **Priority** relations such that the first member of each tuple is the same. This corresponds to the equality join of relational database systems.

It is important to note that inquiries are existentially quantified. An inquiry is evaluated as "if there exists a tuple **<x1, ..., xn>** in relation R, return **true.**"

It is also important to note that the logical variable binding done during pattern-matching is in effect only for the duration of the rule. The scope of a logical variable, then, is the rule where the variable occurs.

Other variables may have more permanent bindings, and behave like constants. There is no syntactic distinction between free and bound variables. To avoid confusion in examples, free logical variables will always begin with a lower case letter, bound variables and constants will begin in upper case.

Another type of antecedent condition is a test for the absence of a tuple. This condition has the form

```
if ¬TapeDrives(j, 4) -> . . .
```

which is read "if it is not the case that a tuple **<j, 4>** exists in the **TapeDrives** relation." If the tuple pattern is not a member of the relation, the expression is evaluated as **true.**

The absence of a tuple from a relation might or might not be interpreted as the negation of its presence. If one uses a "closed world" assumption, then absence is the same as negation. The logical interpretation of an absence test is dependent on the programmer's intent and assumptions.

Free variables are not bound in an absence test. Consider again the rule:

```
if ¬TapeDrives(j, 4) -> . . .
```

For the antecedent to be true, there is no tuple <j, 4> in the TapeDrives relation. Therefore, the free variable j will remain unbound.

The inquiry and absence conditions form the basis for the evaluation of the current state of the system. An additional mechanism is provided for the evaluation of state information. This mechanism is termed a constraint, and can be any Boolean expression.

Consider the case where one is interested in determining if a job requires more than 5 tape drives. This could be expressed as :

```
if TapeDrives(j, n), n > 5 -> . . .
```

where the expression n > 5 is a constraint. The join example shown previously could be rewritten as

```
if TapeDrives(j1, n), Priority(j2, p), j1=j2 -> . . .
```

The consequent portions of rules alter the state of the system. The actions of the consequent typically update relations in some way. The fundamental actions are assertions and deletions.

An assertion adds a tuple to a relation. Consider the rule:

```
if Request(resource, job), Avail(resource) ->
     Allocate(resource, job).
```

If the contents of the Request and Avail relations match the inquiry patterns, the rule will "fire", and add the tuple <resource, job> to the Allocate relation.

In the previous example, the free variables resource and job were bound in the antecedent portion of the rule. These bindings were maintained through the consequent portion of the rule and determined the instantiation of objects added to the Allocate relation. Free variables, therefore, must be bound through pattern-matching before their use in the consequent of a rule.

The allocation example raises some problems. The rule successfully allocates a resource to a job by the assertion to the Allocate relation. This assertion, however, does not alter the conditions Request and Avail that initiated the rule's firing. Thus, this rule may conceptually "fire forever" unless some action is taken to disable one or more of its conditions.

The deletion is used for this purpose. The allocation rule may be written as:

```
if Request(resource, job), Avail(resource) ->
     ¬Request(resource, job),
     ¬Avail(resource),
     Allocate(resource, job).
```

The deletions ¬Request(resource, job) and ¬Avail(resource) remove the indicated tuples from their relations.

The preceding actions--determine a pattern-match of a tuple within a relation, then remove the tuple--is a typical sequence. An abbreviated syntax for this sequence is the cancel operation. Using cancel operations, the preceding rule may be written:

```
if *Request(resource, job), *Avail(resource) ->
     Allocate(resource, job).
```

The cancel operation returns true if the indicated pattern is matched against a tuple in a relation. If the antecedent of the rule succeeds (all conditions are true), then the tuples matched during cancel operations are deleted from their relations.

Rules may be coupled through alternation. In the preceding rule, an alternate action may be desired if the requested resource is not available. This is expressed as:

```
if *Request(resource, job), *Avail(resource) ->
     Allocate(resource, job)
else if *Request(resource, job) ->
     Blocked(resource, job).
```

The antecedent of the alternate rule will be evaluated if the primary rule fails. In this example, the effect will be to place the job in a blocked state.

E. THE APPLICATIVE COMPONENT

Function application is used to support the state transitions described above. In those cases where a tuple must be specified, an applicative expression may be used to compute the value of a member. Consider the following rule:

```
if *TapeDrives(j, n), n + 1 <= 10 ->
     TapeDrives(j, 2 * n).
```

This rule uses infix arithmetic operators to compute values in a constraint and during an assertion. Such operators are permissible in constraint expressions and in the consequent portions of the rule, with the restriction that variables participating in such expressions must be bound. In the preceding example, the variable n was bound in the TapeDrives inquiry.

Named functions may also be used within expressions in the same way as the infix operators. This is shown in the following rule:

```
if *AvailTapeDrives(L1), *TapeQueue(L2),
   (l1 ¬= Nil) & (l2 ¬= Nil)  ->
     AvailTapeDrives(Rest[L1]),
     TapeQueue(Rest[l2]),
     Allocate(First[L1], First[l2]).
```

In this example, available tape drives and jobs queued for tape drives are represented as lists. The functions First[x] and Rest[x] return the head and tail pointers of their argument.

Functions are declared as follows:

```
fn fact[x] : if x <= 1 -> 1
             else x * fact[x-1].
```

This example illustrates that function bodies are similar to rules, and are in fact conditional expressions. The antecedent of a conditional expression is a Boolean expression, but not an inquiry or absence test. The consequent of a conditional expression is another expression, not an assertion or deletion. Thus, conditional expressions (and function bodies derived from them) are free of side effects. As the factorial example illustrates, functions may be declared recursively. Iterative constructs are not defined.

F. PROCEDURES

A typical invocation sequence for a rule begins when a tuple is added to a relation. The tuple is associated with an agent--the object or process that made the assertion--and the agent often expects a group of rules to execute as a result of this assertion. Finally, the agent may expect a value or object to be returned.

To illustrate this, consider the assertion of a tuple in the Request relation. Such a tuple may be <a, r>, where the object a is a relation representing the agent, and the object r indicates the resource desired. This assertion was made to trigger the following rule:

```
if *Request(a, r), *Avail(r, id) ->
    a(id).
```

In this example, the relation a is used as a mailbox to receive a response from the server rule. The assertion of the tuple <a, r> is similar to the creation of a conventional activation record, and the mailbox a is similar to the activation record of the caller.

The assertion a(id) places the desired response--a resource identifier--in the relation belonging to the requesting agent. When this response appears in the mailbox relation, the requesting agent may extract the result and continue its computations. The mailbox relation serves as a synchronization and value-returning mechanism.

In an event-driven system, such a calling sequence would be a common usage pattern. This is recognized by the inclusion of a calling mechanism within the language. The above sequence could be initiated by a synchronous call. Consider the following rule:

```
if *InitProc(p), *Require(p, r), ¬Allocate(p, x) ->
    Allocate(p, Request{r}).
```

The Request expression in this example is a synchronous procedure call. Its effect is the automation of the mailbox handling of the previcus rule. The call Request{r} will be translated by the system into an assertion Request(a, r). The object a is a system-supplied relation that will receive a response from rules firing as a result of the assertion. When a tuple is added to this relation, the tuple is

returned as the result of the procedure call in the expression where the call was invoked.

The procedure call as shown in the preceding example is similar to the function invocation described earlier. Both invocations may be used in expressions, returning results that are incorporated in expressions. The underlying mechanisms of the function and procedure call are different, however. In particular, the procedure call relies on the use of rules to describe its actions, and therefore relies on side effects.

The server rule in this example may be triggered by either an assertion or procedure call involving the Request relation. There is nothing about the form of the rule that indicates its use in procedure calls. By convention, however, such rules must use the leftmost member of a tuple in the enabling relation as the receiver of the response.

The value returned by a procedure call does not have to be used in an expression. In the following example:

```
if *Function(job, c), ¬CodeTable(c, def)  ->
        Display{"Illegal function code"}
```

an assertion to the **Display** relation is assumed to eventually cause the message to be displayed at the user's terminal. The value returned by the calling mechanism is used for synchronization only, and is otherwise ignored.

G. SEQUENTIAL CONTROL

In the preceding examples, no particular order was assumed for the evaluation of conditions in the antecedent, and no order is assumed for the execution of the actions in the consequent. These actions may be considered to be asynchronous and concurrent. This situation becomes even more unstructured when a collection of rules is being

considered for evaluation. Once again, no evaluation order is assumed.

The sequential block provides a mechanism for the programmer to specify an explicit order for rule evaluation. This is shown in the following:

```
if *Request(a, r), ¬Avail(r) ->
     {  -> Display{"Waiting for resource..."};
        if *Queue(r, l) ->
             Queue(r, cons[a, l]);
     }
```

The effect of this rule is to display a message and to add the requesting agent to a queue for the desired resource. The sequential block guarantees that the rules within the {}'s will be evaluated in the order shown.

As the example indicates, the variables bound in the antecedent retain their bindings in the sequential block. The blocks may be nested, with the bindings of free variables extending to inner blocks.

In this example, the antecedent is omitted in the Display rule. This is equivalent to a true antecedent. When writing such rules, the notation may be shortened to:

```
Display{x}
```

which is equivalent to:

```
if TRUE -> Display{x}.
```

H. CONTROLLING THE NAME SPACE

The previous descriptions give a simple mechanism for the binding of logical variables. The rules may be summarized as:

- The scope of variables bound in an antecedent extends to the consequent for a given rule.
- If sequential blocks are nested, bindings made in outer blocks extend to inner blocks.

The previous examples have suggested a more global binding mechanism, as indicated in the use of relation names. The Request relation name is globally bound in this manner. The mechanism through which global names are managed is the directory.

A directory is a named collection of pairs, <name, definition>, where the name entry is a character string and the definition is an object or value representation associated with the name. The directory may be thought of as a named symbol table.

MacLennan [Ref. 14: pp. 34-35] describes a directory structure with two partitions: public and private. Names defined in the public partition are globally visible to agents other than the owner. Names defined in the private partition are visible only to the owner.

A simple elaboration of this mechanism provides a flexible partitioning scheme. The public and private partitions are associated with named directories. These directories are organized into named classes, not necessarily disjoint. In addition, there is a notion of a "current directory" similar to that in UNIX.

The context of a name, then, is determined by the current directory in which the name is defined. When evaluating the binding of a name, the current directory public and private partitions are searched for an entry. If this local search fails, the public directories associated with the classes of the current directory are searched. The class structure defines a search path for variable lookup.

To illustrate these points, consider the definition of the Request relation of previcus examples. Suppose the current directory is "ServerDatabase," a member of the class "Servers." The relation is created and named by:

```
Define{Private, "Request", Newrel{}}.
```

The Define procedure call makes a <name, definition> entry into a directory partition. The Newrel{} procedure call is assumed to return a unique system identifier that represents a relation. The definition shcwn, therefore, would create the relation and bind a name to that relation in the private partition of the current directcry.

Such a server relation would be of more general utility than a private definition allows. Before the relation is opened to broader access, an access control mechanism is necessary.

This control is achieved by associating capabilities with each relation. When a relation is created with the Newrel{} procedure call, full capabilities are associated with the relation identifier. These capabilities include read, add, and delete. A public definition of the Request relation may be accomplished as follows:

```
Define{Public, "Request", AddOnly{Request}}.
```

The AddOnly procedure call references the system identifier, with full capabilities, that has been bound to the private name Request. A copy of this identifier is made with reduced capabilities but still referring to the same relation. This new identifier is then installed in the public directory for general access. This technique of capability addressing is based on the wcrk of Dennis and Van Horn [Ref. 17].

Objects are created in a similar manner:

```
Define{Private, "Job1", Newobj{}}.
```

The Newobj{} procedure returns a unique identifier to be associated with an object. Objects created in this manner have no intrinsic value associated with them, and there is no access control associated with their identifiers.

I. A PROGRAMMING SYSTEM

The language elements described may be adapted to a general programming system. To simplify interaction with the system, [Ref. 14: p. 39] suggests the use of rules as a command language.

While syntactically similar to production rules, these command rules are subject to a slightly different method of interpretation. If a user wishes to query the contents of the Allocate relation, this may be accomplished by:

```
if Allocate(x) -> Display{x}.
```

If analyzed as a production rule, however, this query would be a "fire forever" type. Rules such as this require a different method of evaluation: test, fire, and forget.

The command rules represent the second class of rules in the system. The first class of rules is that of the production rules previously described. These rules are termed active rules. Active rules comprise a body of state transition information that continucusly monitors the relations referenced in their antecedents. Command rules are initiated by an event in the system, and only evaluated once. The initiating event in previous example was the entry of a command rule at the terminal.

The active rules are distinct from command rules, yet the command rules provide the interactive interface between the user and the system. The two categories are bridged with the rule denotation.

A rule denotation is a syntactic representation of rules as data. A potential active rule may be described:

```
Define{Private, "RequestRules",
     <<
     if *Request(a, r), *Avail(r) ->
          Allocate(a, r), a(r)
     >>
}.
```

The denotation is expressed between << >>'s, which is interpreted as "parse but don't evaluate." This definition binds the parse tree associated with the server rule to the name "RequestRules" in the private directory partition.

A rule denotation bound in this manner is a data structure subject to manipulation by the system. To make the transition from this passive status to active status, the rule denotation is activated:

```
Activate{ServerRules}.
```

At this point, the rules expressed in the denotation are moved to active status and enter a continuous test-fire cycle.

This process is similar in many respects to program development in a more conventional system. The command entry of the rule corresponds to the creation of a program source file, and activation corresponds to compilation, linking, and loading.

III. DESIGN ISSUES AND GOALS

A. THE ARCHITECTURE OF RULE-BASED SYSTEMS

Omega is a production-rule system. Davis [Ref. 18: p. 301] describes these systems in terms of three components:

- A rule base. Omega's set of active rules.
- A database. The set of relations and their contents.
- An interpreter. The mechanism for rule selection and execution.

B. RULE SELECTION AND CONFLICT

The control cycle of the interpreter processes rules in a continual recognize/act cycle. The recognition phase consists of selection and conflict resolution [Ref. 18: p. 325].

Omega uses a forward-chaining method for rule selection. This method, described in the examples of the previous chapter, compares the antecedent of the rule to the database. A rule is selected when an appropriate match is found.

In general terms, production systems produce a conflict set for each recognize/act cycle [Ref. 18: p. 325]. The conflict set consists of all active rules whose antecedents are true given the current state of the database. In the Omega system, the resolution of the conflict set is simple. For a given cycle, each rule within the conflict set will be tested and, if its conditions are true, will fire. The order in which the rules in the conflict set are tested is not specified.

The rules of Omega are indivisible once they are selected [Ref. 14: pp. 19-20]. To illustrate this point, suppose the following rules were active:

```
if *Request(a) -> Allocate(a, Red).
if *Request(a) -> Allocate(a, Blue).
```

Under this selection strategy, only one of these rules will fire (assuming there is only one tuple in Request). The indivisible nature of rule evaluation guarantees at least mutual exclusion for rules such as these. Since the evaluation order for these rules is not defined, a more explicit antecedent would have to be designed to establish conflict priorities.

The rule selection and conflict strategies support a powerful execution mechanism. Using this approach, the procedural information of the system is sensitive to the state of the database, and responds accordingly. This behavior is more complex than a procedure-oriented system, where the thread of execution control is more closely tied to the procedural code organization.

The complexity of rule testing has performance penalties associated with the precision of rule selection. By precision, we refer to the number of rules whose antecedent conditions are true compared to the number of rules selected for testing. The most inefficient and most obvious level of precision is to scan the entire rule base on every cycle. We call this a global sweep strategy. At the opposite extreme is a selection strategy that produces only "successful" rules for test.

C. PATTERN-MATCHING

At the heart of the rule evaluation process is pattern-matching. Given the form of a rule:

```
if R(e1) -> . . .
```

a description the pattern-matching process for each candidate tuple x in R is:

```
match[x, e1]:
     if x=Nil and e1=Nil
          return TRUE
     else if x=Nil or e1=Nil
          return FALSE
     else if e1 is an atom
          if e1 is unbound
               bind[e1, x]
               history := cons[e1, history]
               return TRUE
          else if e1 = x
               return TRUE
          else
               return FALSE
          endif
     else if match[first[x], first[e1]]
               return match[rest[x], rest[e1]]
     else
               return FALSE
     endif
end match
```

This description assumes a LISP-like list representation for tuples.

This pattern-matching process is expensive. Incorporating this method at the heart of the interpretation cycle presents a significant design challenge.

D. MORE CONVENTIONAL ISSUES

Besides these unusual design issues, Omega is subject to the same design requirements as more conventional languages. These include:

- A parser and lexical scanner for command/rule input.
- A procedure-oriented evaluation component for applicative expressions.
- A flexible symbol table mechanism for the support of directories.
- Dynamic typing.
- Dynamic memory allocation and reclamation.

E. DESIGN GOALS

The design goals for the prototypes are grouped into the areas of feature implementation, relation representations, efficiency, and evaluation.

1. Feature Implementation

The major objective in this work was the construction of working prototype interpreters for the language. A progressive schedule was developed that sought to implement the following features:

- Canonical Rules. This phase includes the development of the inquiry, absence, assertion, and deletion functions for basic rule interpretation.
- Function definition and evaluation.
- Procedure calls.

- Cancel operations.
- Sequential blocks.

2. Relation Representations

Relations and the operations defined on them are the central components of the Omega system. To support flexibility in relations, a variety of representations is desirable. Such representations could range from a simple list structure to a relational database management system (DBMS). To reduce the problems associated with multiple representation, the relation interface must be clear: an abstract data type, with primitive operations defined for data access.

3. Efficiency

The Omega language was developed as a general-purpose language, capable of prototyping programming environments and a variety of system-level functions. This orientation makes execution efficiency an important implementation issue. The goal was to obtain a level of efficiency comparable to a LISP system.

Efficiency was considered mainly in terms of execution speeds. In those cases where a space-for-time tradeoff was available, it was made.

4. Evaluation

The final goal for the prototypes was evaluation. This evaluation centered on performance: how control strategies and data structures affect execution time. A second evaluation area was to determine the utility of language features through programming experience.

IV. A LISP PROTOTYPE

A. WHY LISP?

The design issues for Omega led to the decision to write a quick prototype for the exploration of high-level design decisions. The high-level concerns were the interpreter organization, selection strategies for rules, and the representation of objects, relations, and directories. Efficiency was not a design goal for this prototype.

Franz LISP was selected as the initial prototyping language. This selection was made for the following reasons:

- Availability. Franz LISP was available on the VAX 11/780 system being used for this work. We were familiar with Franz LISP, and the implementation is well-done. The system includes a reliable interpreter, compiler, and debugging package.
- Symbolic facilities aid in pattern-matching. The heart of the Omega design is the pattern-matching process. The symbolic manipulation facilities of LISP allow these algorithms to be programmed quickly.
- Dynamic typing. The dynamic typing of LISP corresponds well with the typing of Omega.
- Memory management. Memory management is transparent under LISP. While these issues can impact heavily on system performance, they are complex and distracting to early prototyping.
- Debugging. The debugging facilities of Franz LISP are excellent, and superior to any other development

environment available at the time. In a prototyping project, extensive debugging is essential to cope with constant design and coding changes.

B. ORGANIZATION

The interpreter is organized like a classic LISP interpreter. The organization is based on the description given in Chapter 11 of [Ref. 19].

The top level consists of a read-evaluate-sweep loop. The read function is a command rule parser. Commands are entered at the terminal, parsed, and an instruction list is generated. This instruction list is passed to the evaluation function for execution. The sweep function processes any active rules that are ready to fire after the actions of the read-evaluate phases are complete.

C. THE LEXICAL SCANNER AND PARSER

The reader consists of a lexical scanner and parser. Instead of evaluating input as LISP expressions, the reader accepts free-format input using the Omega grammar.

1. The Lexical Scanner

A character reader function is used to pass a list of characters to the scanner. This reader function uses the character input facility of Franz LISP [Ref. 20: pp. 5.6-5.7]. As each character is read, it is added to an input character list. The complete character list is passed to the scanner.

The scanner processes the input character list to recognize tokens. When recognized, a character list is compressed into a token (LISP atom) using the implode function [Ref. 20: p. 2.11]. The final output of the scanner is

a list of tokens built up in this manner. It is this complete list of tokens that is passed to the parser.

The token classes consist of identifiers, constants, and delimiters. Constants are limited to integers and strings. Once a constant token is recognized and constructed, a denotation function transforms the symbol into a LISP integer or string atom.

The separation of the scanner from its supporting reader function allows the same routines to read from multiple sources. Two reader functions are used: one for console input and one for file input. The file input function is used to support command rule entry from text files, similar to the load function of Franz LISP [Ref. 20: p. 5.5].

When receiving console input, the reader needs to distinguish the end of input for a command. Successive carriage returns are recognized as this termination condition.

2. The Parser

A single-pass, recursive descent parser is used to process the token list produced by the scanner. As a construct is recognized, an operator symbol is created which, along with its operands, is added to the parser's output list. The parser receives a token list as input, and returns an operator/operand list as output.

The output list for the parser is a simplification of the abstract syntax for the language. Consider the following input:

```
if *R1(x), R2(x) -> R3(x).
```

This rule is reduced to a token list by the scanner and input to the parser. The parser output for this rule would be the following LISP expression:

```
(
    (CANCEL  (INQUIRY (VAR "R1") (TUPLE (VAR "x"))))
    (PRESENT (INQUIRY (VAR "R2") (TUPLE (VAR "x"))))
    (ASSERT  (VAR "R3") (TUPLE (VAR "x")))
)
```

The sublists preceded with the symbols CANCEL, PRESENT, and ASSERT are termed instructions. A list of these instructions is produced for every rule.

These instructions correspond to the basic actions that the interpreter must perform. Besides the three instructions shown, this prototype produces similar instructions for the deny operation (DENY).

Subordinate instructions are created for operand generation and evaluation. In the preceding example, the sublists headed by INQUIRY, TUPLE and VAR fall into this category. Other subordinate instructions are included for constants (CON), rule denotations (DENO), and function application (APL).

A subset of the original Omega grammar is recognized by the parser. The intent was to allow the interpreter to evaluate the simplest canonical form of rules, which uses present/inquiry tests, absence tests, assertions and denials.

The Omega grammar described in [Ref. 14] makes no syntactic distinction between the antecedent and consequent of a rule. Thus, a rule would be written:

```
R1(x), R2(x) -> R3(x), R4(x).
```

The convention of allowing the "->" to be omitted in a command rule requires an indeterminate lookahead to decide whether the antecedent or consequent portion of the rule is being parsed. If the "->" is omitted, every token in the input list must be examined.

This problem is solved by a pre-scan of the token list before parsing. If the token list doesn't contain the "->" token, it is inserted at the beginning of the list.

D. RULE EVALUATION

The interpretation of a rule is done by an iterative execution function and a subordinate recursive evaluation function. The execution function steps through the instructions in a rule and passes them to the evaluation function. This separation into iterative and recursive functions is done to facilitate backtracking.

The evaluation function returns a value of "true" or "false." If an instruction is evaluated "false," the execution routine resets any temporary bindings associated with that instruction's predecessor, then attempts to re-execute the predecessor. The rule fails when this process backs up to the initial instruction. It succeeds if all instructions succeed.

At the level of the execution function, there is no distinction between the conditions of the antecedent and the actions of the consequent. Backtracking is only meaningful, however, in the antecedent portion of the rule. Therefore, the evaluation function always returns "true" for instructions generated from the consequent of a rule unless an error is detected.

An instruction history list (a stack) is maintained to support backtracking. If a backtrack is required, the pointer to the predecessor instruction is popped from this list. If the instruction succeeds, the pointer for the instruction is pushed on the list.

A binding history list (another stack) records the logical variable bindings being made as each instruction is evaluated. If an instruction fails, the bindings of its

predecessor are popped from the binding history list and undone. If an instruction succeeds, any free variable bindings made during its evaluation are pushed on the binding history list.

The cancel operation requires the generation of a DELETE instruction should the cancel evaluate as "true." A delete list is maintained for this purpose. To support backtracking, the delete list is checked for duplicates before adding instructions. The instructions in the delete list are passed to the evaluation function after all the instructions for the rule have successfully executed.

1. Instruction Evaluation

The instruction evaluation function performs a direct interpretation of the instructions produced by the parser. The steps of the function are simple:

```
Eval[ l ]:
        return Op [ Eval[ o1, o2, . . ., on ] ]
        where Op is the operator of l and
        <o1, . . ., on> are the operands of l
end Eval.
```

The function recursively evaluates the operands of an instruction, then applies the instruction's operator to the result. To support this organization, a LISP function is defined for each operator.

2. The Applicative Component

The applicative component of Omega is simply supported by function definitions in LISP. Such functions are invoked by the APL operator, which is passed the function name and its arguments. These symbols are directly interpreted by LISP, and a result produced.

This applicative mechanism bypasses some important issues which a non-LISP implementation must consider. These issues include function definition at the rule level and the interface between the object-oriented and applicative interpreter mechanisms.

The simplicity with which new functions can be defined to support the Omega interpreter gives this implementation a strong reliance on functions. Consider the implementation of a rule for **Display**:

```
if *Display(x) ->
     Null(Print[x]).
```

The function call **Print**[x] is translated directly to the LISP print function. **Null** is a dummy relation whose only purpose is to allow the print function to execute.

In the above example, the use of the **Print** function is inconsistent with the philosophy behind the applicative component of Omega. The LISP **print** results in a side effect, and is therefore not a pure applicative expression.

The **print** mechanism is more accurately modeled through relations. A functional implementation is forced in situations such as this to allow access to LISP definitions. The consequence of this "bending" of the semantics of the language is shown by the appearance of awkward constructs such as the **Null** relation.

3. Objects

An object in Omega is used as a place-holder in relations. To fill this role, the fundamental characteristic of objects is uniqueness. This was easily implemented using the **gensym** function of Franz LISP [Ref. 20: p. 2.8], which returns a unique symbol each time it is called. A function to create new object identifiers needs only to make successive calls to gensym.

4. Relations

Relations are represented as objects which have an associated list of tuples. The relation object identifiers are created through the gensym mechanism previously described. A value is bound to the symbol using the set function of LISP.

To illustrate this representation, consider the following sequence of command rules:

```
Define(root, "R1", Newrel[ ]).
R1("a", "b", "c").
R2("x", "y", "z").
```

The **Newrel[ ]** function returns the object identifier for the new relation. After the assertions, the relation **R1** consists of the following:

```
(
     ("a" "b" "c")
     ("x" "y" "z")
)
```

The relation is a list of tuples, each of which is a list.

With the list representation for relations is a set of management routines. These routines are **match**, **add**, and delete.

The match function provides the mechanism for pattern-matched inquiries. The function is constructed as follows:

```
match[RelationName, Tuple, Index] :
     if Index = Nil
          R := get binding of Relation Name.
     else
          R := Index
     endif
     while R <> Nil
```

```
            if match_equal[first[R], Tuple]
                return R
            else R := rest[R]
        end while
        return FALSE
    end match
```

The **match_equal** function is a modified version of a recursive list equality test. A modification is required for unbound variables. The algorithm is:

```
    match_equal[R1, R2] :
        if R1=Nil and R2=Nil
            return TRUE
        else if R1=Nil or R2=Nil
            return FALSE
        else if R1 is UNBOUND
            return TRUE
        else if R1 is an atom
            return R1=R2
        else if match_equal[first[R1], first[R2]]
            return match_equal[rest[R1], rest[R2]]
        else
            return FALSE.
    end match_equal.
```

The **match** function performs a linear search of the relation. Each tuple is selected and tested until a match is found or the list of tuples is expended. The index parameter passed to the **match** function indicates the point in the relation where the search is to begin. This indicator is non-nil when the match is being requested on a backtrack attempt.

The **match_equal** algorithm is similar to the pattern-matching algorithm discussed in the previous chapter. In

match_equal, however, no variable binding is done. Instead, variable bindings are made after match_equal returns its result.

The add function places new tuples at the beginning of the relation list, making the relation list a LIFO structure. The delete function removes a tuple directly from the relation list.

5. Binding

Variable binding is controlled by symbol tables for temporary and global bindings. These symbol tables are implemented as association lists, and are manipulated through the following management routines:

- Add. Install a symbol and its definition in the symbol table.
- Delete. Remove a symbol and its definition.
- Lookup. Given a symbol, search the table and return the definition.

The use of association lists for symbol table representation is not the most efficient method provided by LISP, but does offer some advantages. The representation is simple, and the table contents can be easily inspected during debugging. Also, there is a close correspondence between these association lists and the structure chosen for relations.

The correspondence between symbol tables and relations allows the direct implementation of directories as relations. The directory provides an environment which binds variables during rule evaluation.

To support the use of multiple directories, the rule structure was expanded to include an environment pointer for each rule. This environment pointer represents the

directory to be used for variable lookups. A rule, then, is represented as a pair: <ep, ip>, with environment pointer (ep) and an instruction pointer (ip). This representation is called a closure, and is a technique used to simulate static variable binding in LISP systems [Ref. 19: pp. 436-37].

To support temporary bindings made by pattern-matching, a local symbol table is used. The evaluation of variable bindings follows the following sequence:

- When a rule begins execution, a global environment pointer is set to the environment pointer for the rule.
- To evaluate a variable, the local symbol table is searched for a previous definition. If not already defined, the directory referenced by the environment pointer is searched for a global binding. If globally bound, the variable and its definition are installed in the local symbol table.
- If not defined in the local symbol table or in the rule's directory, the variable is considered unbound. This is represented by the installation of a special "unbound" definition in the local symbol table.

Variable binding details are external to the relation management functions. Before passing a tuple to the match function, lookups are made in the local symbol table and variables replaced by their definitions. The match function accepts this tuple as input, and returns a pointer to the corresponding relation tuple if a match is found. Free variables become bound by having their match counterparts added as definitions in the local symbol table.

The isolation of the relation match function from variable binding simplifies the interface between the relation management routines and the evaluation function. This simplistic approach is flawed, however.

Consider the following rule:

```
if *R1(x, x, y) -> R2(x, y).
```

Assume the relation R1 only contains the tuple <1, 2, 3>. Using the match algorithm previcusly described, the pattern <x, x, y> would successfully match against <1, 2, 3>. To prevent such an error, the match algorithm must take temporary bindings into consideration. This requires an exposure of some of the details of the binding mechanism to the relation management routines.

E. CONTROL

Active rule interpretation occurs during the sweep phase of the interpreter's top level. This prototype uses the simplest possible control strategy for rule selection: each active rule is tested on every cycle. Active rules are maintained in a list, and the execution function is mapped to each of the rules. In LISP terms, this is written:

```
(mapcar '(lambda (rule)
                 (exec (car rule) (cadr rule)))
         ActiveRuleList))
```

The lambda function splits each active rule into its instruction and environment components.

After each cycle, the above sequence returns a list of results from each application of the execution function. The result for a cycle appears as:

```
(t t nil nil t nil . . .nil)
```

where each "t" response comes from a successful rule execution. The sweep phase will continue to cycle through the active rule list until all rules return a "nil" response. At this point, the active rule list is in a quiescent state,

and additional cycles will not produce any new state information.

F. ERROR CONDITIONS

Binding requirements differ as rule evaluation proceeds from the antecedent instructions of the rule to the consequent instructions. These requirements constitute a significant source of error.

In the antecedent, the members of a tuple may or may not be bound. An unbound variable at this point is not an error, unless the variable is involved in an applicative expression. In the consequent of a rule, all variables must be bound. Unbound variables at this point are reported as an error.

The binding for relation names is more strict. Given a left-to-right evaluation of instructions, each relation name must be bound before its evaluation.

Consider the rule:

```
if *R(x), x(y, R) -> . . .
```

The evaluation of relation R occurs first. The variable R must be globally bound, or an error will occur. In contrast, the variable x is bound in the R(x) inquiry. Its later use as a relation name is valid.

The requirement that relation names be bound is an implementation restriction. A more general mechanism would allow a sequential search of all relations in the database for trial bindings of relation names. This would extend the free variable binding process previously shown only for the tuples in a relation.

A simple error-handling approach is used in this system. When an error is detected, a message is displayed and the interpreter continues with the evaluation of the next instruction.

G. A BOOT SYSTEM

After the implementation of the basic rule interpreter, it was necessary to identify a minimum set of definitions to support a working system. When such a system becomes operational, additional features can be added through rules defined in Omega.

The foundation of the naming mechanism is the **Define** relation. No rules, relations, or constructs may be added to the system without the use of **Define**. To accommodate names that are added as the system grows, a minimum of a single directory is necessary.

In this prototype, a root directory, defined in LISP, contains the initial bindings required by the interpreter. The root directory initially contains the bindings for the **Define** relation and the active rule list. This directory also contains a reference to itself: a binding for the name "root."

To support the definition of system functions in LISP, the names of these functions are pre-defined at the time of system initialization.

The initial root directory appears as:

```
(setq root '(
     ("Root" root)
     ("ActiveRules" ActiveRules)
     ("Cons" cons)
     ("First" car)
     ("Rest"  cdr)
     ("Append" append)
)
```

This association list binds the Omega names to the appropriate LISP symbols.

When the interpreter begins execution, the following events occur:

- The root directory is loaded into LISP.
- An Omega initialization file is parsed and evaluated.
- The interpreter begins its read-evaluate-sweep cycle.

The initialization file contains Omega command rules that allow the implementation of system functions with rules. These rules are defined by the following assertions:

```
Root("Activate", Newrel[ ]).
ActiveRules(Parse[Fread["sysgen.rul"]]).
```

The assertion to the ActiveRules relation initializes the system's set of active rules to those contained in the file "sysgen.rul." These initialization rules consist of:

```
if *Define(dir, name, def) -> dir(name, def).
if *Activate(newrules), *ActiveRules(oldrules) ->
     ActiveRules(Append[oldrules, newrules]).
```

These rules and definitions are sufficient to set up a minimum system. As shown by the rules for **Define** and **Activate**, it is possible to express system functions as rules. To expand these functions, more rules may be defined and added to the active rule list.

H. LESSONS LEARNED

This early prototype was instructive, both in those functions which worked well and in those functions which performed poorly.

The major benefit was the implementation of a top-down design. The read-evaluate-sweep cycle demonstrated that a recursive, LISP-like interpreter design was useful for Omega.

The prototype implemented a simple list representation for relations, and assisted in the identification of primitive operations required to manipulate relations. Of particular interest was the pattern-matching algorithm. While the implementation was flawed, the basic algorithm was useful in the follow-on prototype.

The iterative backtracking algorithm was more complex than necessary. The stacks used to support backtracking suggested a recursive algorithm as a possible alternative.

The design chosen for the parser was a poor one. The steps of creating a character list, then a token list, and finally an instruction list, were time consuming. The requirement to scan the token list for the presence of the "->" token worsened the already poor performance of the parser.

The interpreter used the crudest possible control strategy, and tested every rule on each iteration of the sweep cycle. This control strategy has the obvious advantage of simplicity, but the performance is unacceptable. The control strategy, together with the slow parsing speed, resulted in a sluggish system response, even with a small number of active rules. In one test case, the parser required 13 seconds to process a 33 line rule file; with an active rule list of about 20 rules, a simple **Display** command took 2 seconds to execute.

It was anticipated that the performance of this prototype would be poor, and so it was. This is not a reflection of LISP as an implementation language. No attempt was made to write efficient LISP, and substantial improvements can probably be made. Potential areas for improvement are:

- An improved parser. The character i/o in Franz LISP lends itself to the inefficient implementation used in the prototype. A possible improvement would be a

scanner and parser written in C, and integrated into Franz LISP as a foreign function [Ref. 20: pp. 8.4-8.8].

- Improved control strategies. More precise rule selection strategies impact heavily on performance.
- More efficient LISP. Franz LISP offers alternatives to the simple list structures used in this prototype. An analysis of the prototype performance could be performed to pinpoint areas for LISP code optimization.

The LISP prototype was intended to be a throw-away implementation. While numerous improvements are possible in this prototype, the performance of the LISP interpreter becomes a final limitation. An implementation in a lower-level language offers the potential for data structures, i/o facilities, and memory management techniques that are more closely tuned to the requirements of Omega.

An important decision in the life of a throw-away prototype is when to stop. This prototype was abandoned after the implementation of a limited but fundamental set of features. The prototype was revised numerous times, but with a minimal expense in coding time and implementation complexity. While many aspects of the interpreter design changed in the follow-on implementation, the contributions of this prototype to the next were substantial.

V. A FOLLOW-ON IMPLEMENTATION IN C

A. WHY C?

The second prototype was written using the C language, although other alternatives were available. The decision to to use C was based on the following:

- High level control structures. The language has a reasonable set of control structures that support modular programming.
- Simple but flexible data structuring. C supports a limited but flexible set of data types and constructors that are well-suited for interpreter implementations. The bit-level operations and weak data typing provide opportunities for space and speed optimizations.
- Recursion. C is a recursive language, and many of the algorithms explored in the LISP prototype were easily translated into recursive C versions.
- Integration with UNIX. As with the LISP prototype, the follow-on was developed on a VAX-11/780, using Berkeley UNIX (BSD 4.2). No language is better suited to UNIX than C, and vice versa. The operating system provides many features that directly support access to system routines and variables. The numerous software development tools available on a UNIX system are largely intended for use by C programmers.

C is not a perfect implementation language by any means. The availability of low-level cperations and type coercion provide a dangerous source of error and confusion. The terse syntax is difficult to read for those unfamiliar with

the language. Finally, C's strict use of call-by-value forces a proliferation of pointer usage, complete with a flood of subtle errors resulting from pointer abuse.

Despite its limitations, C is a tool well-suited to its environment.

B. CHANGES TO SYNTAX AND SEMANTICS

1. An Antecedent Keyword

The previous chapter discussed the problem caused by the optional "->" sign in the Omega syntax. The problem was circumvented in this implementation by the use of the keyword "if" to signify the beginning of the antecedent of a rule. A one token lookahead is sufficient to detect this.

The original Omega syntax uses the "if" keyword to signify a constraint. Thus a rule would be written:

```
*R1(x), *R2(y), if x > y -> . . .
```

Syntactically, the keyword is not necessary to distinguish a constraint. Therefore, the use of the keyword was modified to solve the lookahead problem. Using this modification, the rule is written:

```
if *R1(x), *R2(y), x > y -> . . .
```

The semantics are unchanged.

2. Rule Denotations

The delimiters for a rule denotation were originally asymmetric quotes. This was modified to the <<. . .>> construct shown in previous chapters. This syntax was selected to add greater visual emphasis for rule denotations.

3. Rule Separators

A small but important modification was made to the use of the period and comma as delimiters. MacLennan uses the semi-colon to separate rules within a sequential block, and a period is used for the separation of rules in a denotation. This distinction is made to emphasize the sequential nature of the block in comparison to the concurrent nature of the rules within the rule denotation [Ref. 14: p. 23].

This distinction was altered to solve the command rule termination problem. Rules are always separated by semicolons; a period indicates the end of the current command rule input. This replaces the dual carriage return termination of the earlier protctype.

This problem results from the use of rules as a command language. Rules tend to span multiple lines, so a simple end-of-line termination is not sufficient to indicate the end of input. Two alternative solutions to this problem are: (1) terminate a multi-line command with a continuation character, or (2) use a special character to signify the end of input.

The latter technique was selected, with some loss of the useful syntactic distinction between denotations and sequential blocks. It is hoped that the remaining distinctions between the two constructs, <<>>'s vs. {}'s, are sufficiently different to serve as a reminder of the differences in semantics.

4. Parameter Lists

The original syntax for a function call was similar to the form of an assertion or an inquiry. Thus a rule would appear as:

```
*R1(x, y) -> R2(cons(x, y)).
```

The square bracket notation for function parameter lists was selected to provide an obvious distinction between function calls and other non-applicative forms in the language. The square brackets also denote lists, with the similarity emphasizing the semantic connection between these constructs.

A similar modification was made for the procedure call, where curly braces denote the parameter lists. This syntax appears unusual, and the procedure call mechanism is unusual. The semantic similarities between the procedure call and the sequential block, both of which provide some degree of control on the otherwise free concurrency of Omega, is emphasized by their common use of curly braces.

5. Conditional Expressions and Function Definitions

The introduction of an applicative component into the language required syntactic extensions. These extensions were centered around the conditional expression, which is illustrated by the following rule:

```
if *R1(x) -> R2( if x<3 -> "YES" else "NO" ).
```

The value of the assertion is determined by the conditional expression.

Given the form of the conditional expression, a function declaration can be formed by giving the function name, parameter list, and body (a conditional expression):

```
fn Max[x, y] : if x > y -> x else y.
```

Syntactically, a function declaration may appear anywhere a command rule would appear.

The form of the conditional expression is a modification to the original syntax of the rule, with restrictions to prevent side effects.

6. An Implicit Response for Command Rules

A syntactic change was made to allow expressions at the same level as assertions. This modification allows the entry of the following command rule:

```
if *R1(x) -> x.
```

When this rule is evaluated, the binding of x is "returned" by the rule. If this binding is printed by the interpreter, the necessity for ubiquitous "Display" calls may be lessened. This allows the command entry of expressions such as:

```
2 + 2 * Sin[Pi/2].
```

where the interpreter returns the result.

While this modification to the rule syntax is convenient for command rules, it provides some interesting semantic questions. Suppose the following is an active rule:

```
if *R1(x) -> 2 + 2.
```

What does the consequent of this rule mean? It involves no alteration of the database, but instead requires an expression evaluation.

The action may be described by the following equivalent form:

```
if *R1(x) -> Eval{"2 + 2"}.
```

The expression is asserted to an implicit Eval relation, and the semantics of the procedure call apply. Note that, in general, the value returned by a procedure call used at this level is ignored. For command rules, this value may be used to indicate the result returned from evaluating a rule.

Given this interpretation, consider the following rule:

```
if *R1(x) -> R2(x).
```

What is the "result" returned by the assertion? A simple convention is that that the assertion of a tuple x to a relation R returns the tuple x as its result.

7. Head/Tail Pattern Specifications

A final added feature is a head/tail pattern specification for lists, similar to that of Prolog [Ref. 21: p. 43]. This is shown in the following rule:

```
if *R1([h:t]) -> R1(h), R2(t).
```

The [h:t] notation will match a list. The variable h will be bound to the head (first) of the list, the variable t will be bound to the tail (rest) of the list.

The head/tail specification syntax is extended for tuples:

```
if *R1(h:t) -> R1(h), R2(t).
```

This notation provides a pattern specification for tuples that is independent of tuple cardinality. This generality was not possible using previous constructs.

C. DATA STRUCTURES

Data structures posed the major design challenges for this interpreter. The structures of particular interest were the representations for rules and for supporting the objects and values of the language.

1. A Uniform, Tagged List Structure

A list structure, similar to that of LISP, was selected for the representation of rules. This structure was selected for the following reasons:

- Rules are represented as binary trees. Using a list structure for rules allows a direct, recursive evaluation technique similar to that of the LISP prototype.
- Omega needs lists. Lists provide a general constructor mechanism that is extremely flexible. In a pattern-matching language, list structures are essential if pattern-matching is to extend beyond character strings.
- Uniform list structures simplify design. Given that lists are desirable as a data type within the language, a simple set of list handling routines suffices for analysis and synthesis of data within the interpreter. Uniform list structure also allows storage allocation and reclamation to concentrate on a single unit: the list cell.

A diagram of the basic cell structure is shown in Figure 5.1 The cell has three fields: a tag field, a head field, and a tail field. Table I shows the types of values that each field may assume.

The atomic values in this implementation are character strings, signed integers, and objects. These atoms are represented by cells. The type of an atom, as with all cells, is determined by the value of its tag field.

Integers have their values contained directly in the head field of a cell. Likewise, objects have their identifiers encoded in this field. The width of this field is 32 bits, determined by the VAX 11/780 word size.

Character strings have a pointer in the head field that references a contiguous block of string storage. Reflecting their C implementation, string storage areas are NULL terminated. On the VAX, NULL is represented by a zero value.

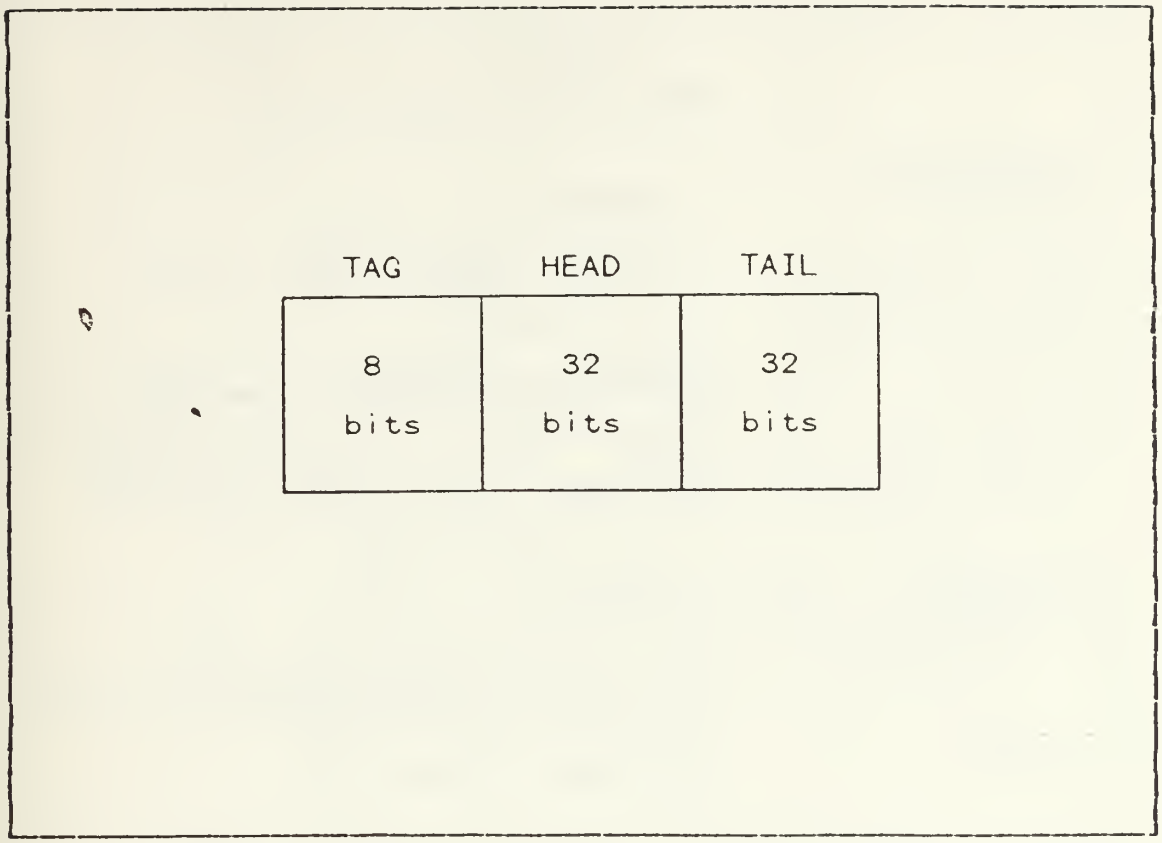

Figure 5.1 Cell Structure.

A list cell contains pointers in both the head and tail fields. There are two classes of list cells: data list cells and operator list cells. These are distinguished by tag values.

Data list cells are analogous to LISP lists. They serve as data constructors. Operator list cells form the interior nodes of a binary tree representation for rules. Rules are transformed and evaluated as tree structures. The "instruction" concept of the prototype was dropped in favor of a more uniform approach. Figure 5.2 illustrates the parse tree for a simple arithmetic expression.

This tagged structure simplifies evaluation at the expense of storage space. Individual tags are used for all

TABLE I

Cell Field Values

head field: Contents	Comments
cell pointer	data and operator list cells
string pointer	string cell
integer	integer cell value used for frame size in allocate op
object id	object cell--id is 32 bit integer
<block,offset>	VAR cell--gives scope and offset within binding stack frame
tail field: Contents	**Comments**
cell pointer	data and operator list cells VAR cells and defined objects have pointers to print names
unused	integer, string, and and most object cells

primitive data types and for each construct (node) in the abstract syntax for rules. This results in a large number of tags: over 40 in the current implementation. A minimum of 6 bits, therefore, is required to represent the tag of a node. Given the cell space requirements given in Figure 5.1, approximately 11 percent of the system storage requirement is needed for tags. (The actual percentage is somewhat less because of character string blocks and hash tables, discussed below).

2. Objects

As in the LISP prototype, objects are represented by a unique identifier. In this implementation a cell is

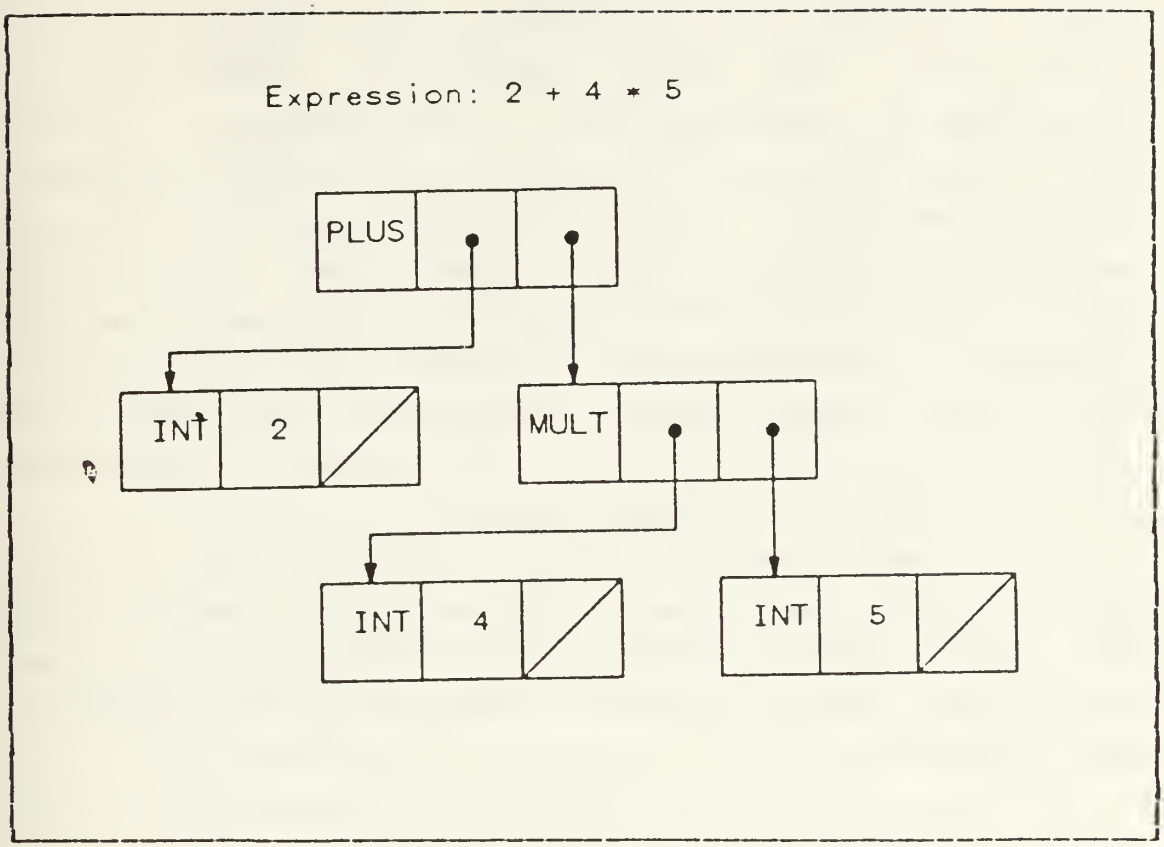

Figure 5.2 Tree Representation for a Simple Expression.

generated for an object and ar identifier embedded in the head field. Again the problem is how to manage the identifiers so that the are unique.

The approach used is tc maintain an object count. Each time a new object is requested, the object count is incremented. If the control of object identifier allocation remains centralized, the objects are guaranteed tc be unique.

The generation of identifiers this way leads to the issue of object identifier management. What prevents the system from running out of unique identifiers? What happens when the identifier space is exhausted?

A simple strategy is to ignore these problems altogether. How long can the system generate identifiers before it runs out? For a rough calculation, assume that a new identifier is generated every 10 milliseconds. The head field which contains an identifier is 32 bits, so assume 29 bits are available (the use for the remaining 3 bits will be discussed shortly). With these values, the identifier space would be exhausted in 2^{29} x 10 milliseconds, or about 62 days. The management of object identifiers is not a major issue in this system. Should a larger object identifier space be needed, additional bits could be provided from the tail field of the object cell.

A small portion of the object identifier space is reserved for system use. In the current implementation, the first 64 object identifiers are reserved. The presence of a system object is easily detected by an examination of its identifier.

3. Hash Tables

As in the LISP prototype, certain types of objects have values associated with them. These values are managed in this implementation using a uniform hash table mechanism.

The hash table index is generated by a simple hash function. The algorithm is based on that given in [Ref. 22: p. 135]. The hash function receives a pointer to a cell as an argument, and returns the table index. The algorithm is as follows:

```
hash[p]:
    if p@ is an integer or object cell
        return p@head mod TABLESIZE
    else if p@ is a string cell
        return sum[p@head] mod TABLESIZE
            where sum[s] returns the sum of
                the ASCII characters in the
```

```
                    string s
        else
            return 0
        endif
    end hash.
```

This function is a crude hashing algorithm for string entries, with its primary virtue being simplicity. Object identifiers are generated linearly, however. The direct hash off these identifiers should result in minimal collisions.

The structure itself consists of a pointer table (an array), with a collision list maintained for each entry. The collision list links together a collection of header cells. These cells have a key field with a list cell pointer, a definition field with another list cell pointer, and a link field with a pointer to the next header cell. Figure 5.3 illustrates this structure.

To complete the hash table description, a collection of management and access routines are used. These are:

- Lookup. Given a pointer tc a cell, find an entry whose key field points to an equivalent structure. If found, return the definition pointer.
- Install. Add an entry to the hash table. The hash table is searched for an existing entry with the same key value. If found, that entry is replaced. If not found, the new entry is linked into the appropriate collision list. Note that key entries may be any structure: objects, strings, or lists.
- Delete. Remove an entry from the table. The table is searched for the key value. If found, the entry is removed and its collision list is relinked if necessary.

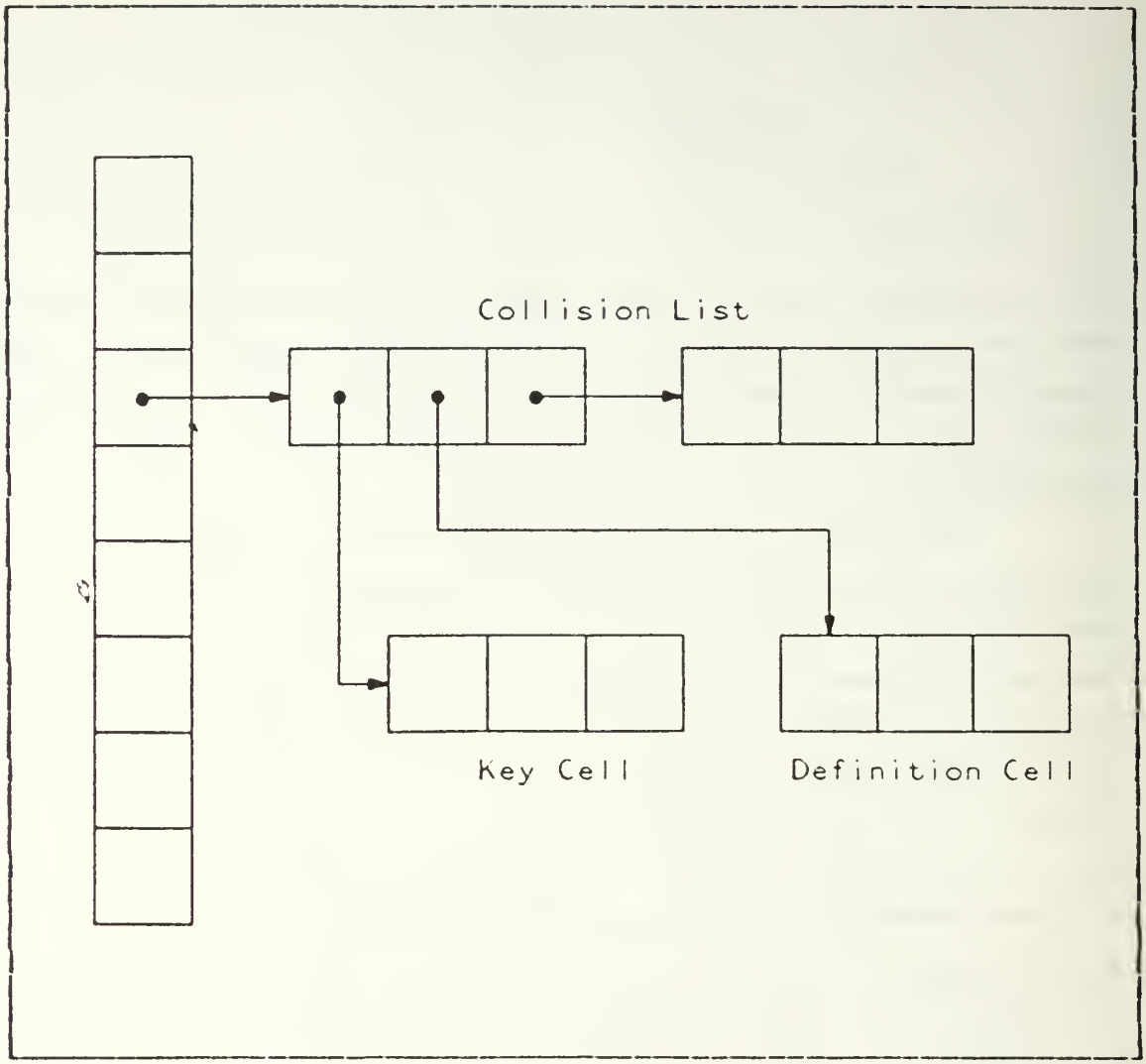

Figure 5.3 Hash Table Structure.

Object representations are linked to object identifiers using these hash tables. The objects within Omega that have representations are relations, directories, rule denotations, and functions. These entities are subject to temporal change, and thus have an object implementation.

4. Relations

Relations are objects, but with a twist: they have access considerations. The access control mechanism is encoded directly into a relation's object identifier. Three bits of the identifier signify whether the relation is accessible for read, add, or delete operations. When a new relation is created, an object identifier is generated and the capability bits all set tc 1, indicating full privileges. Subsequent operations may reduce the capability by copying the relation object identifier and zeroing the appropriate bits. This produces a second reference to the same relation, but with reduced access privileges.

Relations are represented as lists, similar to the LISP prototype. As a tuple is added to a relation, a header cell is created for the tuple and linked in at the head of the existing tuple list (if any). A pointer to this list is bound to the relation's object identifier through the object table. The list representatien of a relation is shown in Figure 5.4

5. Directories

Directories incorporate the hash table into the general list structure. A directory has two header cells: a class link cell and a partition cell.

The class link cell cortains a pointer to a partition cell, and a pointer to the next class link cell in the class. A lookup path, then, may follow this chain from directory to directory.

The head pointer of the partition cell points to the private partition, while the tail pointer of the cell pcints to the public partition. Each partition is represented by a single hash table.

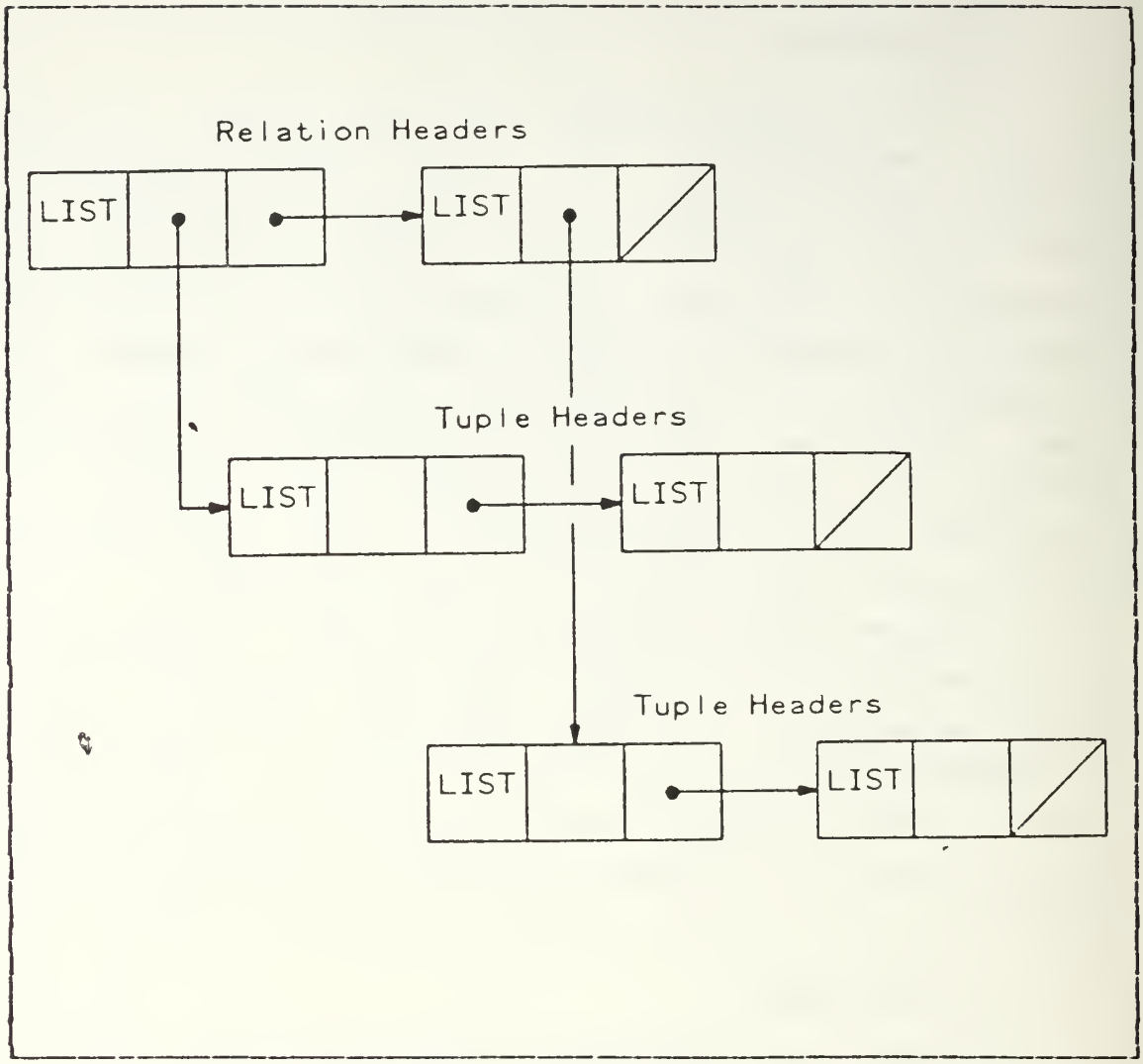

Figure 5.4 List Representation for Relations.

In this implementation, directories are represented differently from relations. This distinction was made to optimize directory access by hashing, although there is a loss of the generality enjoyed by the LISP prototype.

D. ORGANIZATION: THE TOP LEVEL

The interpreter's top level is similar to that of the LISP prototype. An added step--the print phase--results

from the notion that a command rule "returns" a result. The top level, then, consists of a read-evaluate-print-sweep loop.

The read-evaluate-print phases process the user's command entry at the terminal. The print phase provides a visual indicator that some activity is taking place because of the command rule entry.

Suppose a user enters the following rule at the terminal:

```
if *R1(x) -> R2(x), R3(3).
```

The reader parses the expression, binds variables as appropriate, and then passes the parse tree to an evaluation function. The response from the evaluation function is displayed at the terminal. In this example, this response would be "3." When multiple expressions exist in the consequent of a rule, the response from the last expression is displayed as the response for the rule.

After the command rule has been evaluated and its response displayed, the interpreter begins its sweep phase, evaluating any active rules that are ready to fire. There is no implicit response from active rules: their purpose is to alter the database. At the completion of the sweep phase, the command loop returns to the reader and waits for the next entry.

E. THE READER

The reader was a major weakness in the LISP prototype. While an efficient parser implementation was not a major design goal for this work, the slow, error-prone parser of the LISP prototype was frustrating to work with.

A parser generator was used to create the parser in the second prototype. This decision was made with the intent

that a reasonably efficient parser be produced with a minimum of time and effort.

1. A LEX Scanner

The LEX lexical analyzer generator [Ref. 23] was used to produce the code for the scanner. LEX accepts as input a file of rules described through regular expressions and their associated actions. The output from LEX is a table-driven scanner in C source code.

The following sequence defines LEX actions for recognizing unsigned integers:

```
digit      [0-9]
int_con    {digit}+
{int_con}  {
                return(INT_CON);
           }
```

The return statement is an embedded C language construct used to describe the required action by the scanner. In this example, INT_CON is a constant used to represent the token.

LEX is an easy-to-use, sophisticated tool. With no previous experience, we specified, generated, compiled, and debugged a LEX scanner in a few hours. The LEX specification for Omega is contained in Appendix A.

2. A YACC Parser

The parser was written using the YACC (Yet Another Compiler-Compiler) parser generator [Ref. 24]. Like LEX, YACC allows a high level specification for compiler actions. The output from YACC is a table-driven, LALR(1) parser. The YACC parser receives its token input from the LEX scanner.

The following illustrates the YACC specification for an Omega assertion:

```
assertion       :  primary '(' arguments ')'
                       {
                            $$ = newcell(ASSERT, $1, $3);
                       }
```

Additional rules are given for "primary" and "arguments." The **newcell** function generates a new cell with the tag ASSERT, a head pointer set to the value returned by YACC from parsing "primary," and the tail pointer set to the value returned by YACC from parsing "arguments."

The embedded C expression determines the actions of the parser if an assertion is recognized. The assignment to "$$" defines YACC's response: this value is placed on a stack for use in other expressions. In this implementation, the value generated for each rule is a cell pointer. When a form is parsed successfully, the YACC parser returns a pointer to the root of a parse tree constructed this way. The YACC specification for Omega is contained in Appendix A.

YACC is a more complex tool than LEX, and it has some idiosyncrasies. These include precedence specification for infix expressions and a preference for left-recursive grammar rules.

Infix expressions may be specified in YACC by a rule such as:

```
expr : expr OP expr
```

Such a specification is ambiguous, however. To remove this ambiguity, YACC allows the declaration of precedence rules. Thus a precedence rule of:

```
%left '+' '-'
%left '*' '/'
```

would establish the precedence of the arithmetic operators and resolve the ambiguities associated with the previous

rule [Ref. 24: pp. 14-15]. This scheme results in a flat grammar for expressions in YACC, where the precedence rules determine associativity and precedence.

Certain constructs in Omega lend themselves to recursive grammar rules. YACC encourages such rules to be left-recursive. Left-recursive rules result in a smaller parser size, and reduce the likelihood of an internal stack overflow when parsing a long sequence [Ref. 24: p. 19.].

Consider the following, left-recursive specification for a list:

```
list : expression
     | list ',' expression
```

The parse tree generated by such a rule is shown in Figure 5.5 One consequence of this form is that the entire list must be traversed to access the head of the list, an obvious disadvantage in list-oriented interpretation.

To solve this problem while still respecting the YACC preference for left-recursion, a recursive transformation is performed on parse trees. This transformation selectively changes left-recursive forms into right-recursive forms. The algorithm is:

```
rtrans[curr, pred] :
     if curr = Nil
          return Nil
     else if curr points to an atom
          return curr
     else if curr is not a left recursive form
          curr@head := rtrans[ curr@head, Nil ]
          curr@tail := rtrans[ curr@tail, Nil ]
          return curr
     else
          h := rtrans[curr@head, curr]
          t := rtrans[curr@tail, Nil]
```

```
            curr@head := t;
            curr@tail := pred
            if h = Nil
                 return curr
            else
                 return h
            end if
       end if
  end rtrans
```

Figure 5.6 shows the list after the transformation has been applied.

The YACC parser offers several advantages to a prototyping effort:

- Development time. The high level of the specification for YACC minimizes the complexity of parser generation. We had a complete, functional parser working in three days.
- Ease of modification. Experimentation with syntax is simple: change the grammar rules, rerun YACC, and recompile the output. The ease with which the grammar can be modified encourages experimentation.
- Verifying specifications. Analysis of grammar changes is easy in YACC. If a change produces ambiguities, YACC will report conflicts when trying to generate the parser tables. This automated analysis is a strong point in favor of using a YACC parser.

3. Console and File Input

As in the LISP prototype, the same parser is used to read command rules from the console and from text files. This is implemented using the i/o redirection facilities of UNIX and C.

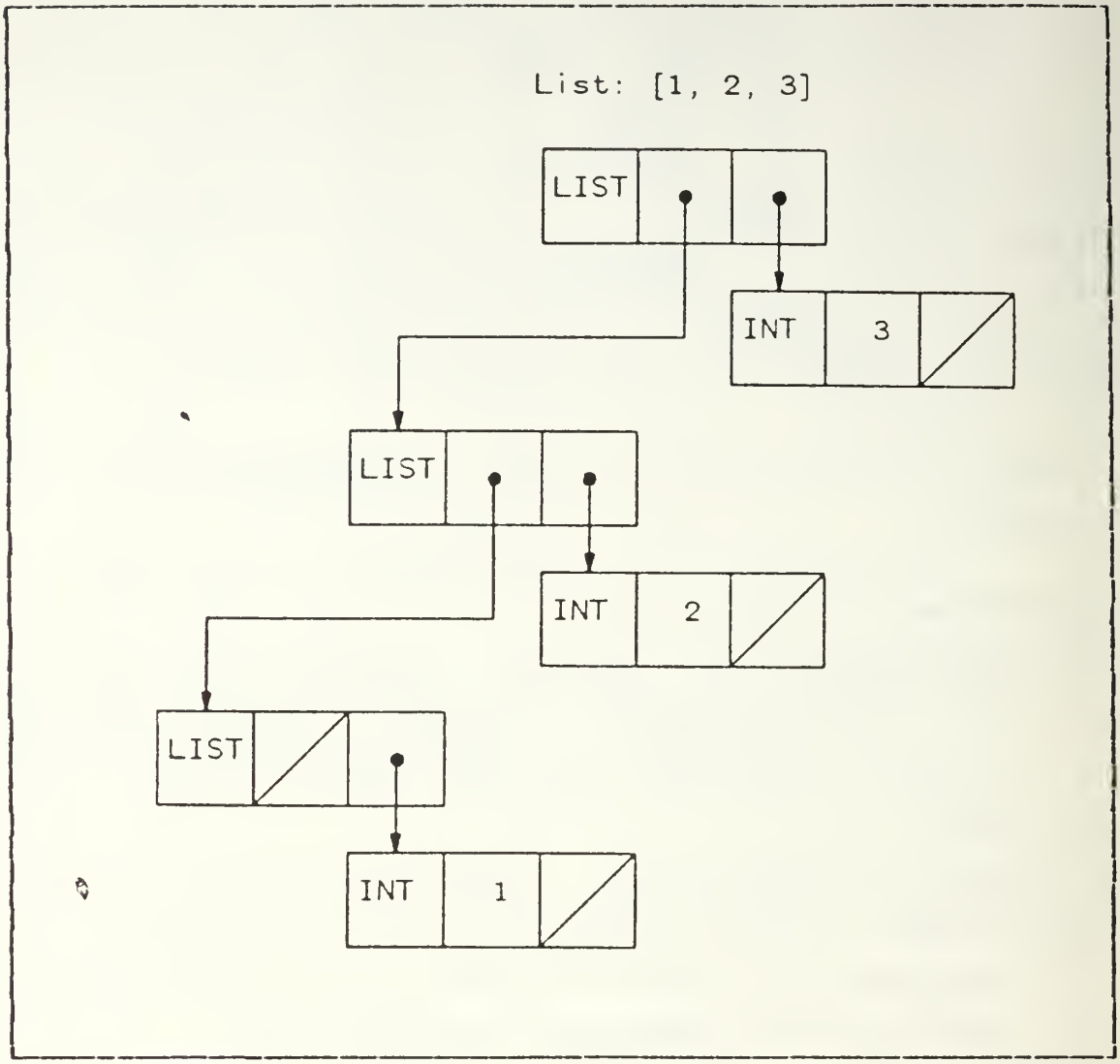

Figure 5.5 Left-Recursive List Representation.

The scanner produced by LEX accepts input from the standard input file by default. To receive input from another text file, the file is opened and the LEX input file variable reassigned. This simple technique allows the alternation of input between several sources.

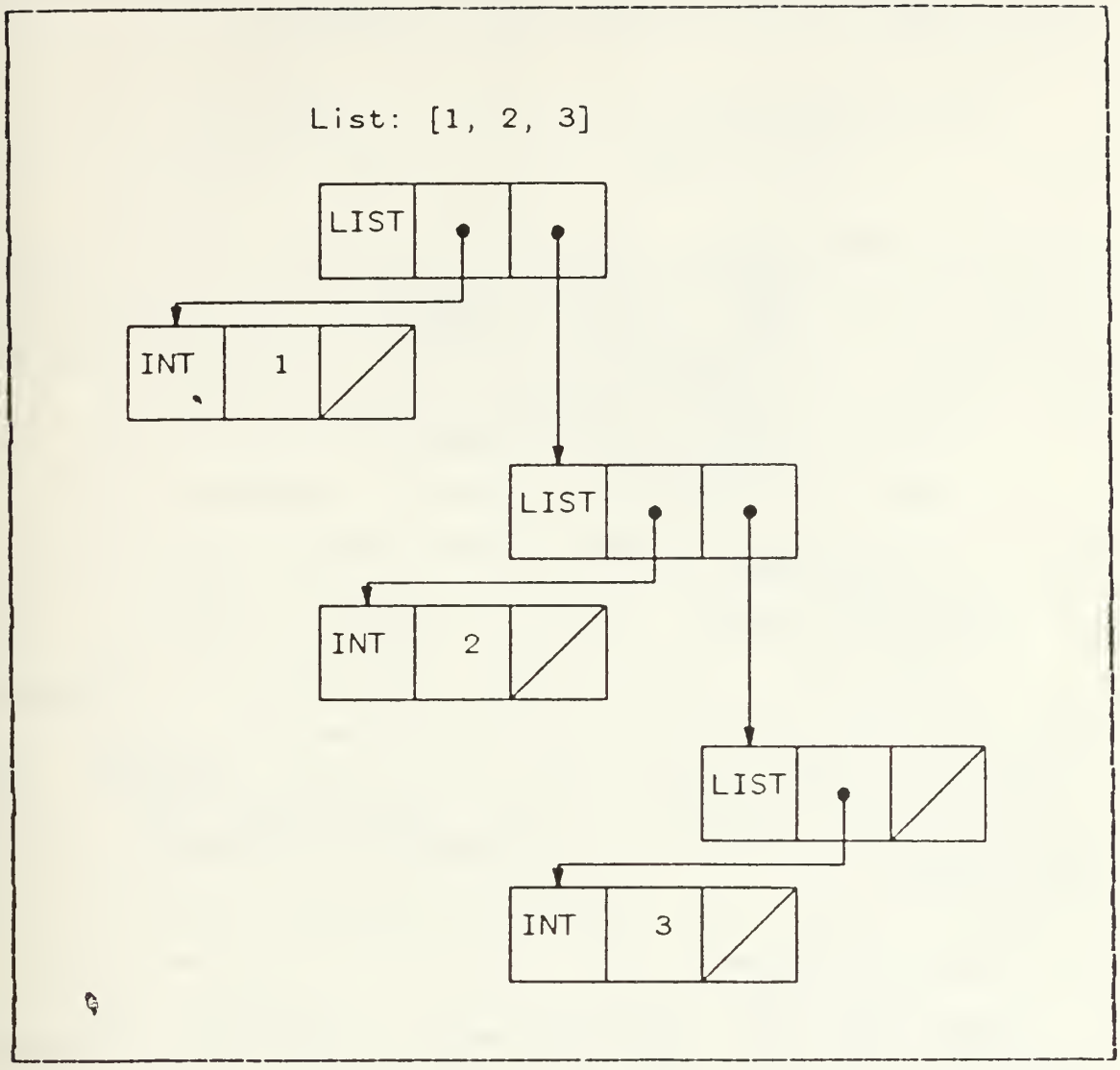

Figure 5.6 Transformed List Structure.

F. A RECURSIVE PRETTY PRINTER

The parser, along with the cell allocation routines to generate the parse tree, was the first system component developed for this implementation. A pretty printer was written at this point primarily to debug the parser.

The pretty printer is based on a large case statement, which selects the appropriate output form based on the tag

of the current subtree. A section of the algorithm may be described as:

```
pretty_print[p] :
     do case p@tag
     . . .
     case RULE :
          print[ "if" ]
          pretty_print[ p@head ]
          print[ "->" ]
          pretty_print[ p@tail ]
          print[";"]
     . . .
     end case
end pretty_print
```

Although the pretty printer originated as a debugging aid, the basic design of the tag-oriented case statement was almost identical for the central evaluation function. This pretty printer evolved into the Display mechanism for the interpreter.

G. RULE EVALUATION

Where the LISP prototype used a separate, iterative execution function for backtracking, the follow-on design uses a recursive backtracking algorithm within a single evaluation function.

As in the pretty printer, the heart of the evaluation function is a large case statement. The tag value of the form being evaluated determines the case selection. A section of this case statement may be described as:

```
eval[ip, ep] :
     do case ip@tag
     . . .
     case RULE:
```

```
            result := rule[ip@head, ip@tail, ep]
        . . .
        end case
        return result
    end eval
```

As in the LISP prototype, separate functions are defined for the majority of interpreter actions. The function **rule** is defined external to the case statement, and contains code for the interpretation of the RULE operator. Separate calls to **eval** are used to evaluate the arguments to **rule**.

The evaluation function receives two arguments: a pointer to the subtree being evaluated (ip), and a pointer to the current directory for global name definitions (ep). The evaluation function returns a cell pointer as its result.

H. BINDING

1. Binding At Activation

This implementation uses an entirely different binding mechanism than the LISP prototype. The variables of a rule denotation are bound when a rule becomes active. These bindings are determined by the environment of activation. Since a command rule is immediately executed, binding takes place immediately for these rules.

The binding process results in a complete copy of the parse tree for a rule, leaving the original denotation unaltered for later use. In this way, the denotation is like a source file, the bound parse tree like an object file.

When a rule denotation is bound, the current directory is searched for variable definitions. The class of the current directory provides a search path to other directories if the variable is not bound in the current directory.

A variable not defined in the directories of the class is a free variable. The cell representing a free variable contains an offset in the head field, and a pointer to the variable's print name (a string cell) in the tail field. The offset for a variable depends on the order in which the free variables of a rule appear.

When a free variable cell is initialized, a pointer to the variable cell is installed as the print name's definition in a local symbol table. Subsequent occurrences of the variable will be replaced by this definition.

During the binding process, the parse trees for embedded rule denotations are installed in the object table. A system-generated object identifier replaces the rule denotation subtrees in their parent expressions. The variables in the rule denotation are left unbound--the binding of these variables is deferred until the denotation is activated.

The final action for the binding process is the creation of an allocation operator cell for the rule. This cell has a count of the total number of free variables for the rule in its head field. The tail field contains a pointer to the actual rule structure.

2. A Binding Stack

The allocation operator is used with a binding stack. The binding stack is an array with a current frame pointer and a chain of dynamic link pointers that connect frames. The binding stack is illustrated in Figure 5.7

When a rule begins interpretation, a stack frame is created on the binding stack with slots allocated for each free variable in the rule. The offset in a variable cell indicates which of the binding frame slots is to be used for that variable.

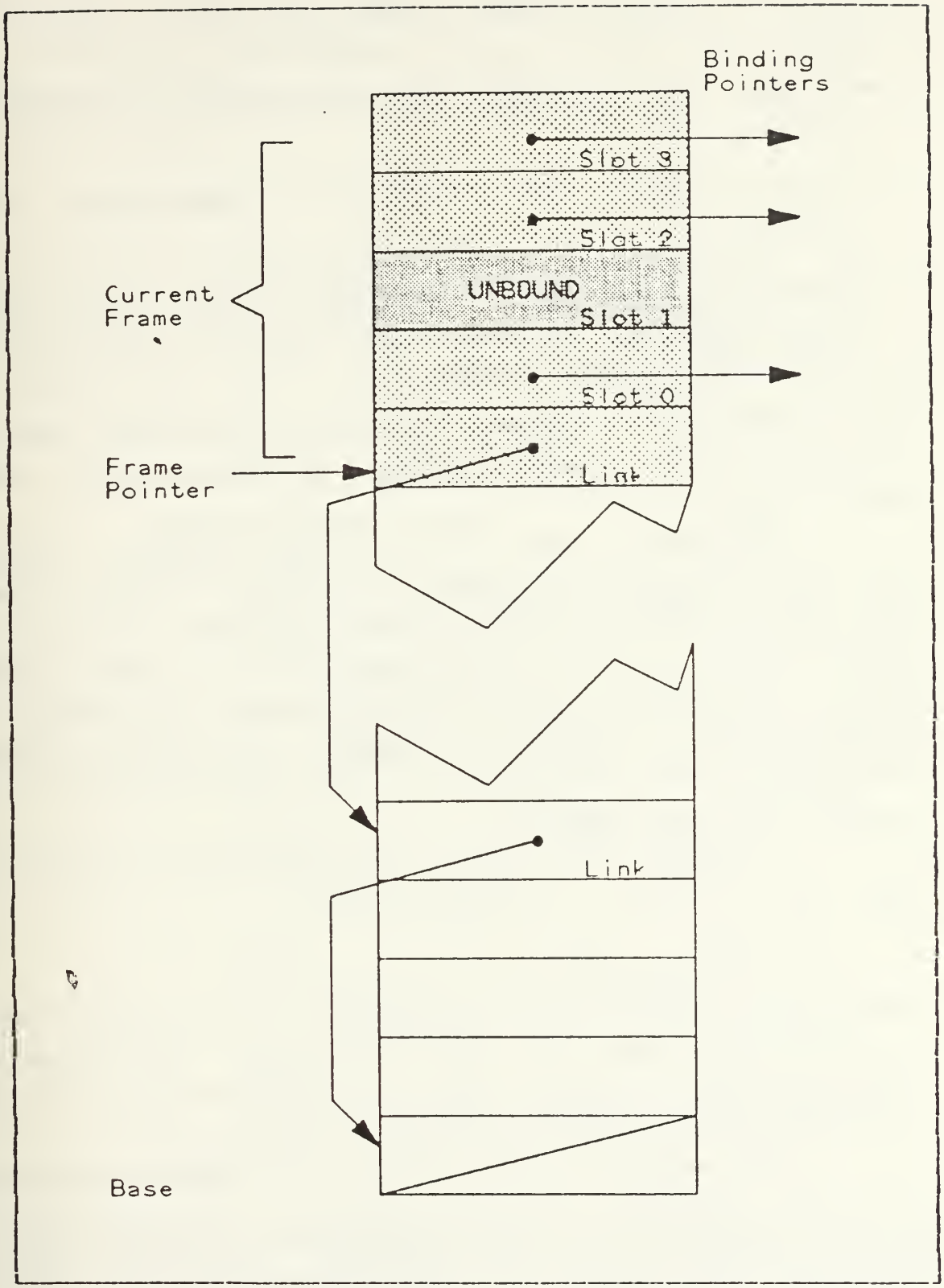

Figure 5.7 The Binding Stack.

The binding process uses the following primitive routines:

- bind[x, y]. The frame slot for variable x is assigned pointer y.
- getbinding[x]. Return the value in the frame slot for variable x.
- freebinding[x]. Free the binding for variable x. This is accomplished by assigning a reserved value to the frame slot meaning UNBOUND.

When a rule completes execution, the dynamic link is followed to the previous frame, and the current frame pointer reset.

The binding stack offers several advantages. First, variable lookups are only done once: at activation time. The dynamic binding process is only concerned with a variable's offset in the stack frame, not with the variable name. The binding stack allows the simple reclamation of storage used for binding. Finally, it allows context switching in rule interpretation since bindings from interrupted rules are preserved.

A context switch for a rule occurs during a synchronous call. Consider the following rule:

```
if *R1(x) -> { R2{x}; R3{x} }.
```

When the R2 procedure call is made, a context switch is made to the body of rules that support the call. The binding of the variable x, however, must be maintained between the R2 procedure call and the R3 procedure call.

This method of static binding eliminates unnecessary variable lookups by replacing variables by their definitions. Generality is still maintained for objects such as relations, whose associated values are determined by dynamic

lookups in the object table where necessary. Thus, the philosophical differences between objects and values in Omega are supported by concrete differences at the implementation level.

I. BACKTRACKING

A recursive backtracking algorithm is implemented with the conditions (CONDS) operator. A condition is an element of the antecedent of a rule: a presence/inquiry test, an absence test, or a constraint. Backtracking is initiated only on the failure of a presence test. The algorithm is:

```
conds[cond_curr, cond_next, ep] :
     match_next_ptr := Nil
     while TRUE
          result := eval[cond_curr, ep]
          if result = FAIL
               return FAIL
          else if cond_next = Nil
               return result
          end if
          result := eval[cond_next, ep]
          if result != FAIL
               return result
          endif
          if cond_curr is not a 'present' op
               return FAIL
          end if
          undo trial bindings made for condition
     end while
end conds
```

This algorithm treats backtracking as a binary operation. The "cond_curr" parameter is the current condition being

tested. The "cond_next" points to the remaining list of conditions to be tested. A successful response from eval on "cond_next" indicates that all remaining conditions have tested successfully. A failure means a backtrack attempt should be made on the current condition.

J. RELATION MANAGEMENT ROUTINES

The relation management routines of the LISP prototype are continued in this implementation. They are: **match**, **add**, and **delete**. As in the LISP prototype, the **add** function links a new tuple at the beginning of the relation list. The **delete** function removes the tuple from the relation list and relinks as necessary.

A pattern-matching algorithm is used in the relation **match** function. As in the LISP prototype, the tuple list of a relation is searched linearly. As each tuple is selected for a match, it is passed to the pattern-matching function. The match function maintains a "match_next" pointer. This indicates where the last match occurred, and provides a search continuation point for backtracking.

The pattern-matching algorithm is similar to that given in Chapter III. Unlike in the LISP prototype, trial variable binding occurs during pattern matching. The pattern-matching function binds free variables by using the **bind** operator of the binding stack. These bindings are undone if a rematch is necessary when backtracking.

In this implementation, relation access control is enforced. The object identifier for the relation is first tested to ensure the capability bit for the desired operation is set. If not, the operation is canceled and an error message is generated.

An additional relation function was added: **match_first**. This function returns a pointer to the first tuple in a relation.

K. ACTIVE RULE PROCESSING

1. Triggers

The technique of triggering is used to improve the precision of active rule processing. The trigger for a rule is the left-most relation in the the rule. For the following rule:

```
if *R1(x), *R2(y) -> . . .
```

the trigger is the relation R1.

A rule is selected for test when certain events take place involving the trigger relation. These events are assertions and deletions. If either of these operations is performed on the trigger relation, there is a likelihood that the rule's antecedent conditions are now satisfied.

The triggering process is initiated at the time a rule is activated. The trigger for a rule is determined, and the rule installed in an active rule table (a hash table), keyed by the object identifier for the trigger relation. A list is maintained in the active rule table for all rules associated with a given trigger.

When an assertion or denial is made to a relation, any rules indexed by that relation are selected from the active rule table and tested.

A rule is always tested at least once: when it is activated. This ensures that any pending conditions will be serviced before the rule enters its triggering cycle.

2. A Rule Queue

Triggered rules are managed through a circular queue. When a rule is triggered, a pointer to the rule is placed in the rule queue.

During the sweep phase, all rules in this queue are tested. If a rule succeeds, it remains in execution by

staying in the rule queue. Instead of undergoing continuous evaluation, a successful rule is reinserted at the end of the queue. This enforces a fairness policy: each rule in the queue should get a turn at evaluation.

Given the nature of rule testing, only one instance of a rule needs to be in the queue at one time. Multiple instances will result in wasted interpreter cycles and excessive queue sizes.

To control this problem, a flag bit is used. The flag bit is contained in the tag field (bit 7) of the first cell in an active rule list. When a rule list is placed in the queue, the flag bit is set. Subsequent attempts to insert the rule list in the queue will be ignored because of the flag bit value. When the rule list leaves the queue (by being selected for testing), the flag bit is reset and subsequent queue requests for the rule will be accepted.

3. Advantages and Disadvantages of Triggering

Rule triggering has the following advantages:

- Precision. The likelihood of triggered rules firing is good. The strategy is much more precise than the global sweep strategy of the LISP prototype.
- Simplicity. The triggering mechanism described is simple, both in concept and in its supporting implementation.
- Triggers are statically determined. The trigger is the left-most relation of a rule. This is a simple, syntactic distinction that is directly inferable from the visual form of a rule.

Despite its attractive aspects, the trigger mechanism just described is too simple. To illustrate this point, consider the following rule:

if R1(x), ¬R2(x) -> R2(x).

The intent of this rule is to enforce the constraint that R1 should remain a subset of R2.

Assume R1 and R2 initially contain the same tuples. If a tuple is removed from R2, the relation contents are different and the constraint rule should fire. Given the previous triggering strategy, however, the rule will not be tested. The affected relation was R2, but the rule is triggered on R1.

4. Two-Level Triggering

A possible alternative to this simple triggering method is to index a rule on every relation in the antecedent. This will guarantee a correct evaluation, but requires a complex index structure. Also, triggering on secondary relations is inefficient--these relations may be updated frequently and result in excessive testing for the rule.

Another possible alternative is to determine the point of failure. In the following rule:

if *R1(x), *R2(y) -> . . .

the R1 inquiry may succeed and the R2 inquiry fail. If the R2 relation is flagged as the point of failure for this rule, a subsequent assertion tc R2 could be the trigger for a retest of the rule.

The difficulty with this strategy is determining the point of failure. Consider the following rule:

if *R1(x), *R2(x, y), *R3(x, z), x > y -> . . .

Each of the relations R1, R2, and R3 may have a tuple that meets the pattern specification. There is a dependency, however, among these inquiries and the constraint. A

failure, then, may be a failure of the combination and not of any particular inquiry.

A compromise strategy is used to solve this problem. We call this technique two-level triggering.

Using two-level triggering, two rule queues are maintained. One is for active, triggered rules selected under the original triggering strategy. This is the primary rule queue. The second queue contains rules pending alteration of one or more conditions to enable firing. This is the secondary rule queue.

When a rule is initially triggered, it is inserted in the primary rule queue. If, when tested, none of the conditions of its antecedent successfully match, the rule is discarded.

If, on the other hand, at least the trigger condition successfully matches (but the combination fails), the rule is entered into the secondary queue. The rules of the secondary queue are tested after the rules of the primary queue have been expended.

Once inserted into the secondary queue, rules will remain under evaluation for possible firing. A rule will leave the secondary queue under two conditions:

- The rule fires and is transferred back to the primary queue.
- The rule fails to match on its trigger relation and leaves the active queues completely.

Two-level triggering may be inefficient. Consider the following rule:

```
if *Employee_Data(name, salary), salary > 10000 ->
     Employee_Data(name, salary/2).
```

If the Employee_Data relation is normally not empty, two-level triggering will perpetually maintain this rule in either the primary or secondary rule queues. If many rules behave this way, the rule selection strategy degrades to a global sweep, a worst-case performance.

Many types of rules fair well under two-level triggering. The key to efficiency for this strategy lies in the use of the trigger relation. This relation should contain matching tuples only when the rule is ready to fire.

L. THE APPLICATIVE COMPONENT

The original description of Omega did not contain a detailed description of the applicative component. Instead, it assumed that a completely separate applicative language, such as MacLennan's A [Ref. 25], would be integrated into the Omega environment to support applicative evaluation.

The applicative component was a minor issue in the LISP prototype, but the follow-on design had to resolve its role and form. Some of the alternatives considered were:

- Develop a general applicative interpreter interface. The Omega interpreter and applicative interpreters would be separate processes communicating through this interface.
- Integrate the code for an existing applicative interpreter into the Omega structure.
- Use simple modifications to Omega grammar and semantics to add an applicative component to the language.

The first option offers the potential for multiple evaluation functions. In one environment, an applicative expression may be evaluated by LISP. Another environment may use an A interpreter.

This option was discarded for efficiency reasons. Applicative expressions use pointers to Omega structures, which implies shared memory access. Separate processes would have to pass this information through an i/o cperation, such as a mailbox transfer or a UNIX socket [Ref. 26].

The second option was discarded because of complexity, and the third selected for the same reason. Minor modifications to Omega itself allowed the rapid development of a simple but useful applicative mechanism.

The only completely new language feature needed was the function definition, which has been shown in previous examples. A function definition takes effect at the same time rule variables are bound.

The function definition performs the following actions:

- The function name is bound to a system-generated object identifier and installed in the current directory.
- The function is separated into a pair, <fp, b>, with formal parameters (fp) and a function body (b).
- The formal parameters are installed as free variables in the local symbol table. The variables of the body are then bound. These variables will contain the stack frame offsets of their corresponding formal parameters.
- An allocation operator is linked to the <fp, b> pair. This operator is used to create space on the binding stack for formal parameter binding.
- The function structure is installed in the object table, keyed on the object identifier.

The function definition is different from the other features of Omega. The mechanism bypasses the "Define" procedure to allow recursive definitions. Note that the installation of the function name and object identifier into

the current directory is done first. When the variables of the function body go through the binding process, recursive references to the function name will be handled properly.

When a function call is evaluated, the function structure is retrieved from the object table. The allocation operator is interpreted, and a frame created on the binding stack for the function's parameters.

The actual parameters for the function call, previously evaluated, are grouped together in a list. This list is traversed, and the pointer for each actual parameter is assigned to a slot in the current binding stack frame. At the completion of this process, all formal parameters are bound.

The function body is then passed to the central evaluation function. At this point, the function body is simply another rule, and it is processed by the rule evaluation routines. The binding stack supports recursion in function evaluation.

This applicative mechanism has the advantages of simplicity and uniformity with the Omega syntax. The function definition, however, does not conform well with the other constructs of the language. Also, lambda expressions and functionals--key components of an applicative language--are not implemented.

Despite its limitations, a variety of interpreter utility functions were defined using this mechanism. These functions are listed in Appendix C.

M. PROCEDURES

With the evaluation and binding mechanisms already introduced, the implementation of a procedure call mechanism is simple. The steps for procedure call evaluation are:

- Evaluate the tuple participating in the procedure call. This tuple is analogous to the actual parameters of a function call, and is implemented as a linked list.
- Generate a new relation object for the mailbox. This object is linked at the beginning of the tuple list.
- Assert the tuple into its target relation. The assertion mechanism will queue any rules triggered as a result.
- Execute the sweep function to evaluate any triggered active rules. The sweep function will continue to execute if there are rules to fire.
- Apply the match_first operation on the mailbox relation to extract the response from the call. It is this response that is returned as the result of the procedure call.

While the procedure call gives a measure of control to rule processing, the mechanism is still unstructured. The philosophy of this implementation is "make the assertion and see what happens." One possible consequence of the mechanism is multiple assertions to the mailbox.

Consider the following active rule:

```
if *R(a) -> a("Yes"), a("No").
```

If triggered as a result of a procedure call, what is the value returned by the call? The use of the match_first operation and the LIFO implementation of relations will return the last assertion as the response. Other assertions are ignored. While this convention seems tractable, it is implementation-dependent.

N. BUILT-IN FUNCTIONS AND PROCEDURES

While implementing the interpreter, the necessity for "hard-wired" functions and procedures became apparent. By hard-wired, we mean that these mechanisms are supported by C functions coded in the interpreter, as opposed to an implementation in Omega rules or functions. These mechanisms are built-in for purposes of efficiency. An example is the **Define** procedure call.

In the LISP prototype, directories were implemented as relations and the **Define** mechanism was implemented with Omega rules. By using different representations for directories and relations, the **Define** mechanism has a different character that requires a more specific implementation.

Names like "Define" are implemented as system objects. Recall that a block of object identifiers is reserved for system use. When a relation identifier is evaluated, system objects are processed by a different set of routines: one for system-defined relations and one for system-defined functions.

The object identifiers for these relations and functions are examined in a case statement, and the appropriate system routine called. The routine for **Define** receives the parameters (pointers) for the target directory, the name, and the definition. The entry is then installed in the hash table for the directory.

The steps required to add a system-defined function are simple:

- An entry is made in the object header file. This file contains the definitions of reserved object identifiers.
- An entry is made in the case statement for the system relation or function handler.

- A directory entry is predefined in the system initialization routine. This routine builds the system's root directory.

This mechanism allows the access of system routines from Omega rules. The procedures for NewRel and NewObj are implemented in this way. Similarly implemented is the Display procedure call, which passes a structure pointer to the system pretty printer.

This system interface replaces the ubiquitous function definitions of the LISP prototype. The process required to implement a feature as a function call or procedure call is the same; the mechanism may be selected that most appropriately models the desired activity. Appendix B lists the built-in functions and procedures for the system.

0. CANCEL OPERATIONS

The implementation of cancel operations relies on two features of the interpreter: the "match_next" pointer into a relation, and the binary backtracking algorithm.

In the backtracking algorithm, a successful evaluation of the "next_condition" pointer indicates that the remaining conditions of the antecedent have all been successfully evaluated. at this point in the recursion, a pointer to the match position in the current relation is available if backtracking is required. If the current operation is a cancel, the "match_next" pointer references the tuple that should be deleted. This deletion is done directly by marking the tag field of the tuple.

The tuple is not removed directly from the relation because a pointer to the tuple's predecessor in the relation list is not available. The alternative to marking is to search the relation from the beginning, maintaining a predecessor pointer, until the canceled tuple is found. The relation could then be properly relinked.

Marking was used to avoid excessive searching of relations. When a relation is scanned on subsequent inquiries, the tuples marked for deletion are removed. In this way, the overhead of linear search is minimized.

MacLennan introduced the cancel operator as a notational convenience [Ref. 14: p. 18]. This simple construct demonstrates several desirable qualities of a language feature:

- The notation is compact, yet readable. The cancel operation removes the necessity to code a redundant delete operation. This saves space in the source file and in the resulting parse tree.
- A potential source of error is removed. The relation name and tuple pattern of delete operations normally correspond exactly to their counterparts in a presence test. It is easy to misspell identifier names in the delete clause.
- Cancel operations allow optimization. The use of the "match_next" pointer reduces search time. When a delete operation is evaluated, there is no easy way to link this to searches conducted when processing the rule's antecedent.

P. SEQUENTIAL BLOCKS

The implementation of the sequential block involves two functional characteristics: (1) the sequential evaluation of rules within the block, and (2) the nested scoping of free variables.

Sequential evaluation is a natural consequence of the interpreter's design. The implementation evaluates the actions of a rule's consequent in a left-to-right sequential order. The rules within a sequential block are processed in the same way.

1. A Single-Pass, Multi-Scope Symbol Table

The nested scopes of sequential blocks require an elaboration of the binding process previously described. Scoping is handled by the following steps:

- A block count is maintained during binding. As a sequential block is entered, this count is incremented. When the binding of the block is complete, the block count is decremented.
- Variables have a block number and an offset. As new free variables are encountered in a sequential block, their offset is determined. The variable index, contained in its head field, now contains two elements: a block number and an offset within the block.

Free variables are installed in a local symbol table as they are encountered in the binding process. To correctly process references to outer blocks, a multi-scope symbol table is required. This symbol table is implemented as a two stack structure: one stack maintains the variable reference pointers, the other stack maintains scope pointers. As each variable is encountered, the symbol table stack is searched from the current stack top to the base. If found, the variable is replaced by the definition returned. New variables are installed in the symbol table by pushing the variable reference on the stack.

As a sequential block is entered, the stack top for the preceding scope is saved on the scope stack. When the binding of the sequential block is complete, the predecessor's stack top is restored from the scope stack. The scope stack partitions the variable reference stack into the appropriate scopes. The structure is illustrated in Figure 5.8. This symbol table structure is similar to structures used for conventional, block-structured languages [Ref. 27: pp. 325-327].

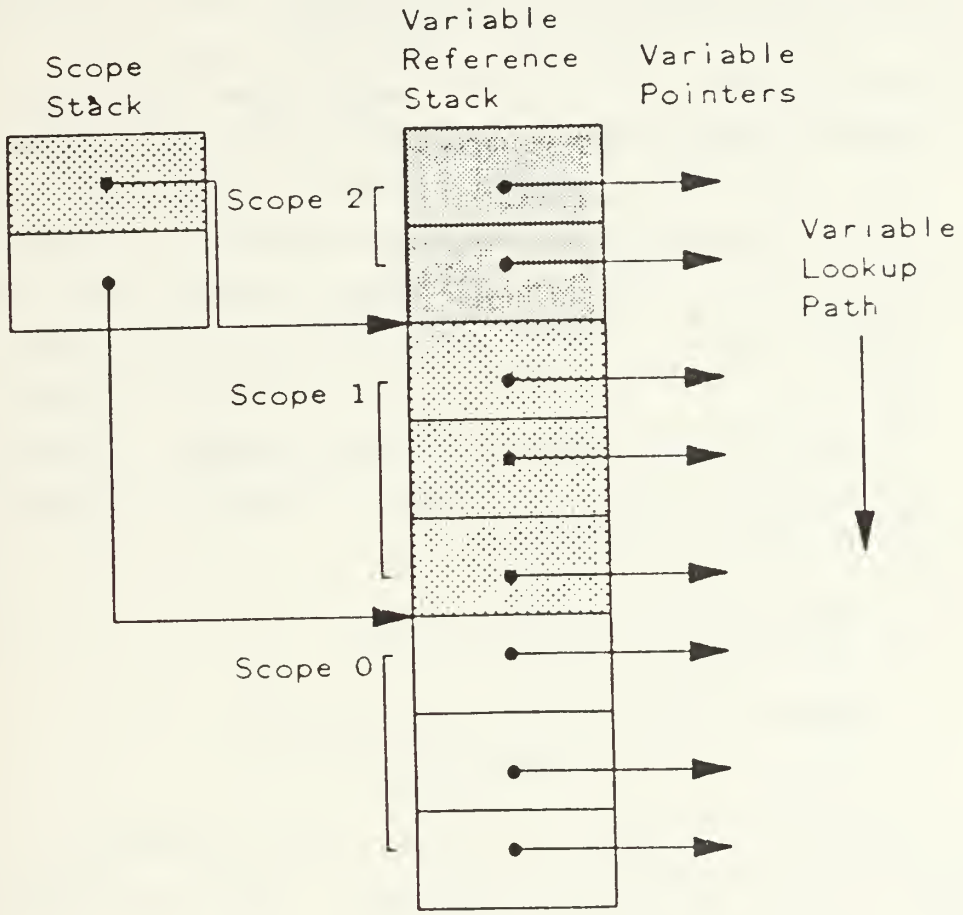

Figure 5.8 Multi-Scope Symbol Table.

2. Evaluation of Multi-Scope Bindings

The dynamic evaluation of bindings requires modification to process the nested scopes of sequential blocks. The evaluation function has the following additional features:

- A global scope count. This is incremented when a sequential block begins execution, and decremented when the sequential block is completed.
- Walking the links. The **getbinding** operation for the binding stack must now take scoping into account. To do this, the block number of the variable is compared to the interpreter's global scope number. If these numbers are not the same, then the correct stack frame is located by traversing n links up the binding stack, where n = variable block number - current scope number.
- Function and procedure context switches. Functions and procedures require new scopes. This is accomplished by the following sequence:

```
Scope_Save := current_Scope
current_Scope := 0
Execute the procedure or function
current_Scope := Scope_Save
```

This process is similar to static link processing in conventional block-structured languages, such as described in [Ref. 19: p. 232-238].

This implementation does not require separate static and dynamic links. Procedures and functions execute in scopes separate from their points of invocation. A single set of links in the binding stack is sufficient to support multi-scope references and the calling chain of functions and procedures.

Q. SYSTEM INITIALIZATION

The system initialization sequence is similar to that of the LISP prototype. The root directory is initialized with the names of the systems's built-in functions and procedures. As in the LISP prototype, the directory has a self-referencing entry.

An Omega initialization file is parsed and evaluated. A call to the **sweep** procedure propagates any rule activity resulting from these rules. This initialization provides the definitions for utility functions and procedures. These utility rules are listed in Appendix C.

To augment the initialization file, the user may specify an Omega file name on the UNIX command line. The interpreter will parse and evaluate these rules as part of the initialization process.

Finally, the interpreter enters the read phase of its read-evaluate-print-sweep cycle.

VI. STORAGE MANAGEMENT

A. THE STORAGE PROBLEM

The basic storage unit for Omega is the cell. These units are allocated dynamically, to support changing list structures, temporary results from computations, and changing relation contents. Dynamic memory allocation and dynamic typing make relation manipulation a flexible but complex activity.

The task of freeing unneeded storage quickly became too complex for explicit memory reclamation in the interpreter design. By explicit memory reclamation, we mean that, at a certain section of the code, it can be determined that a cell is no longer needed and a call to a reclamation routine can be immediately made.

Reclamation is complicated by memory sharing. This sharing is a natural consequence of the design of Omega, and comes from pattern-matching and reuse of active rule structures.

Consider the following rule:

```
if R1(x), ¬R2(x) -> R2(x).
```

When tuples are asserted to the relation R2, two possible strategies may be used:

- Copy the structure. The complete tuple structure is copied, and the copy added to the R2 relation.
- Share the structure. The match operation returns a pointer to the tuple in **R1**. It is this pointer that is bound to the variable **x**, and available for inclusion in **R2**. If the structure is not copied, the pointer added to R2 refers to the same structure as that in **R1**.

Consider another rule:

```
if *R1(1) -> R2([1, 2, 3]).
```

The assertion to R2 adds a tuple generated by a list denotation in the rule -- this denotation is linked into the rule structure. As in the previous example, the assertion mechanism may choose to copy the structure or share the structure.

Structure sharing is preferable for two reasons: space and time. Structure sharing obviously reduces storage requirements by allowing multiple references to the same storage areas. Of more importance in this implementation is a reduction in execution time. The tuples of a relation may be arbitrarily complex list structures. Copying these structures continually is an execution overhead that structure sharing avoids.

B. STORAGE ALLOCATION AND THE UNIX VIRTUAL ADDRESS SPACE

The implementation uses the storage allocator provided in the UNIX C library. The allocation routine is **malloc**. The reclamation routine is free. [Ref. 26]

To understand how these routines work, a description of the UNIX virtual memory map is useful. An executing process has its virtual memory divided into three logical areas: a text segment, a data segment, and a stack segment [Ref. 26].

The text segment contains the program code. This segment is normally shared and re-entrant. The stack segment is used for the system's runtime stack. The stack begins at the highest possible virtual address, and grows down. The stack area is automatically extended as required.

The data segment consists of two sections: initialized and uninitialized storage. The initialized storage area contains statically allocated storage declared in the

program. In this implementation, the binding stack, symbol table stack, and rule queues are implemented as arrays. Their storage allocation appears in the initialized storage area.

The uninitialized storage area is used for dynamic memory allocation. Calls to **malloc** will extend this area. Calls to free will reclaim, compact, and free virtual storage where possible.

The maximum sizes for the stack and data segments are system-dependent and locally tailored to achieve desired performance goals. The system used for this work has the following limits set:

data segment -- 6112 kbytes
stack segment -- 512 kbytes

This organization is illustrated in Figure 6.1

Malloc allocates memory aligned on word boundaries. Structure storage requirements are rounded to the next four byte multiple based on the 32 bit word size of the VAX. The consequence of word alignment is an additional space requirement for cells. Even though the design only specifies 8 bits for the tag field, this requirement is rounded to 32 bits. A cell has 12 bytes allocated, with 3 bytes of storage (25 percent) unused.

C. IGNORING STORAGE MANAGEMENT

The interpreter was initially implemented with no storage management strategy. Cells were allocated when necessary, but no attempt was made to free excess storage.

This policy proved to be unsatisfactory. A lengthy test program exceeds the system data segment limits. On one test, the interpreter ran for 10 minutes before exhausting its available memory. At this point, 384,159 cells had been

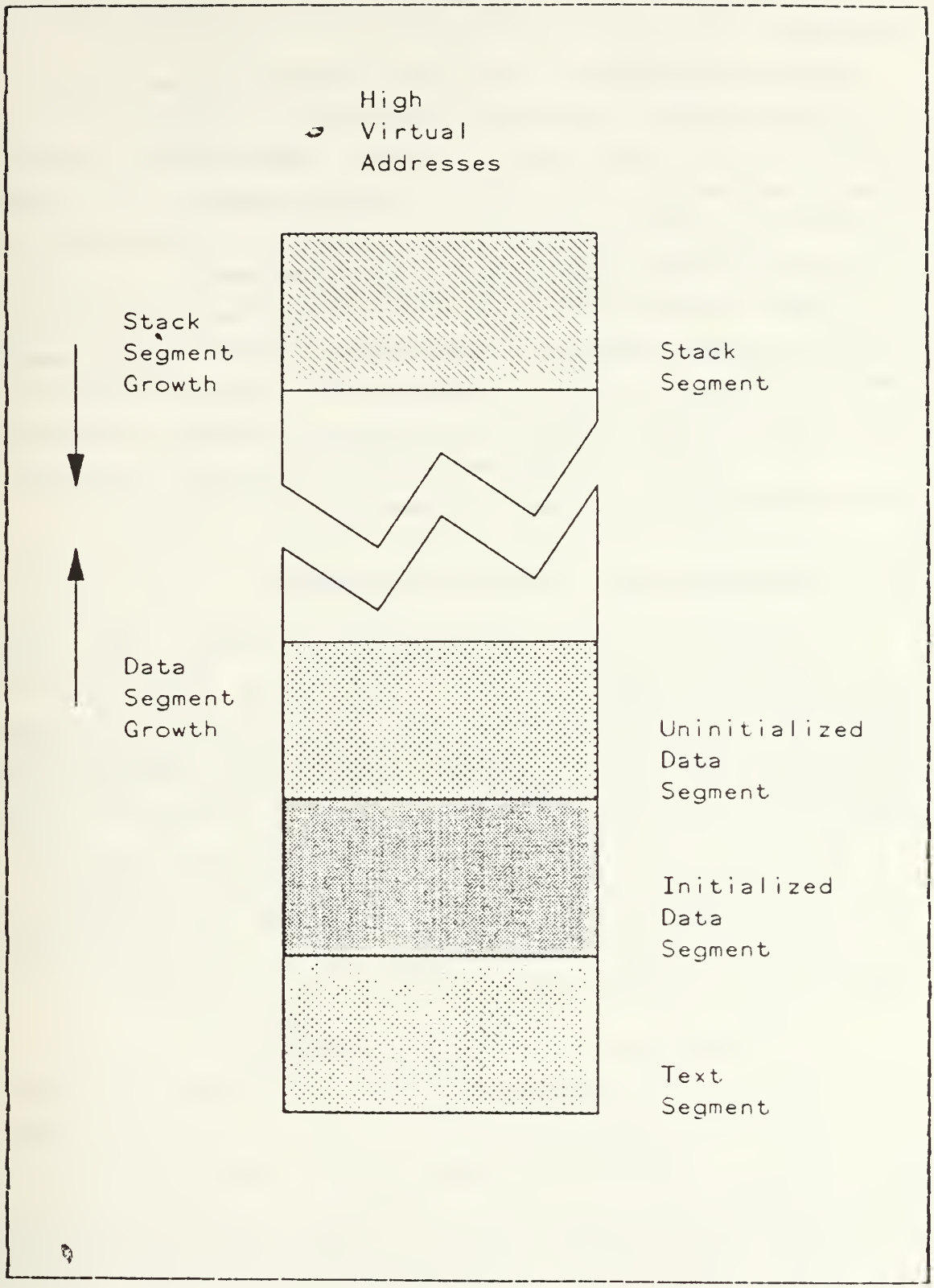

Figure 6.1 UNIX Memory Map.

allocated using 4.5 Mbytes of virtual memory for cell storage.

It is possible, but not necessarily desirable, to increase the data segment limits for a process. The data segment limit used during testing--6 Mbytes--should be more than sufficient for this implementation. A large, constantly growing process also suffers from excessive swap space requirements and a high page fault rate.

These factors point out a familiar lesson: while virtual storage systems allow a large address space, storage management is still a major consideration. These policies are particularly important in a multiuser operating system such as UNIX, where excessively large processes can have an adverse impact on the user community.

D. OMEGA-SPECIFIC STORAGE OPTIMIZATION

Approaches were considered which reduce the storage requirements of the system by focusing on specific characteristics of the implementation. Two areas for optimization were: (1) eliminate expression evaluation during pattern-matching, and (2) reclaim cell storage for the initial tuples added to relations.

The pattern-matching routines are executed frequently as rules are selected for test. An active rule structure is reused, so intermediate results must have separate storage allocated. Consider the following rule:

```
if *IsManager(a, x), Position("Manager: " + x) ->
        a("is Manager")
else if *IsManager(a, x) ->
        a("is not Manager").
```

In the Position inquiry, the "+" represents string concatenation. To evaluate this rule, a new tuple must be

generated to record the results of the concatenation operation. This tuple is then used in the inquiry. Cells such as these may be generated frequently during rule testing.

A possible optimization to this requirement is to restrict expressions in the antecedent of a rule to constraints. Strict pattern-matching is supported primarily by the binding stack and little additional memory is required. If expressions are limited to constraints, and constraints shifted to the end of a list of antecedent conditions, rule failures will occur before the constraint is evaluated, minimizing dynamic memory allocation. Tests conducted using this strategy showed a decrease of cell allocation ranging from 10 percent to 40 percent.

While this strategy reduces memory requirements, the basic issue of storage reclamation is not solved. The restriction on tuple expressions is significant--the programmer must now remember this as an exception to syntax and semantics.

One possible alternative tc the above strategy is the use of a separate storage allocator and reclamation routine for intermediate, temporary storage. This approach was not pursued because a more general solution to the storage management problem was needed.

Certain types of relations tend to be small, with a cardinality of 1 or 0. Consider the following rule:

```
if *Push(a, x, l) ->
        a([x:l]).
```

This rule executes a **Push** operation by **cons**'ing the member x onto the list l.

While multiple agents may have requests to **Push** active at the same time, a typical situation is where **Push** contains a single request which is serviced and promptly removed. A simple strategy optimizes storage for such relations.

A relation list has a collection of cells which we refer to as tuple headers (illustrated in Figure 5.4). These headers link pointers into the relation list which refer to the actual structures of the tuple members. While the pointers may change, the number of header cells is dependent on the tuple cardinality. This cardinality tends to remain fixed after it is dynamically determined.

When the first tuple is added to a relation, the header cell requirements are allocated. A cancel will flag this tuple as deleted by marking the tag, but the tuple will remain linked into the relation list. A subsequent assertion may then reuse these header cells for the next tuple.

This strategy allows a simple optimization of relation storage requirements. Early tests of this strategy indicate a potential 10 percent reduction in cell allocation. The strategy postpones memory exhaustion but doesn't prevent it; a more general storage management policy is still required.

E. REFERENCE COUNTING

Reference counting was selected for cell reclamation. This technique is described extensively in the literature. Our algorithms are based on the material presented in [Ref. 19: pp. 440-442] and [Ref. 28: pp. 383-384].

The implementation of reference counting required the addition of a reference count field in the cell structure and some simple management routines.

Since 25 percent of cell storage is wasted, the inclusion of a reference count field bears no additional cost. The reference count field is implemented as a 16 bit signed integer, although a smaller field would be sufficient.

The reference counting routines are as follows:

- IncrRef -- Increment a cell reference count:

```
IncrRef[p] :
     if p = Nil
          return;
     else
          p@refcount := p@refcount + 1
     endif
end.
```

- DecrRef -- Decrement a cell reference count:

```
DecrRef[p] :
     if p = Nil
          return;
     p@refcount := p@refcount - 1
     if p@refcount <= 0
          if p@ is a string cell
               free string storage
          else if p@ is a list cell
               DecrRef[p@head]
               DecrRef[p@tail]
          endif
          free p
     endif
end DecrRef
```

When a cell for an atom is created, its reference count is initialized to zero. Reference counts are altered when pointer references change. This occurs in the following routines:

- NewCell -- Create a new list cell. The algorithm is:

```
NewCell[x, y]:
        p := malloc[cellsize]
        p@refcount := 0
        p@head := x
        p@tail := y
        IncrRef[x]
        IncrRef[y]
        return p
end NewCell
```

- SetHead -- Change the head pointer for a cell. This is the rplaca function of LISP:

```
SetHead[x, y]:
        IncrRef[y];
        DecrRef[x@head];
        x@head := y
end SetHead
```

- SetTail -- Change the tail pointer for a cell. This the rplacd function of LISP:

```
SetTail[x, y]:
        IncrRef[y]
        DecrRef[x@tail]
        x@tail := y
end SetTail
```

Reference counts also need to reflect the references of a recursive implementation. To illustrate this point, consider the following interpreter routine to implement multiplication:

```
Mult[x, y] :
        IncrRef[x]
        IncrRef[y]
        temp := MakInt[x@head * y@head]
        DecrRef[x]
        DecrRef[y]
        return temp
end Mult.
```

Before its invocation, the arguments to **Mult** have been recursively evaluated. If these parameters involved expressions, then either parameter may be an intermediate result. The **IncrRef** operations reflect the Mult routine's references via its formal parameters. After completion of the multiplication, the references are decremented with **DecrRef**, which will free intermediate results that are no longer required.

Reference counts must remain consistent, so an analysis of local references throughout the interpreter was required. A point of interest is the binding stack: the **bind** operation must increment a cell's reference count while the **unbind** operation must decrement the reference count. The determination of these specific reference counting points proved to be a tedious process, although still more tractable than explicit reclamation.

Reference counting offers the following advantages:

- Simplicity. The data structures and algorithms are supported by the existing recursive interpreter design.
- Immediate reclamation. Unneeded storage is reclaimed immediately.
- Uniform computational requirements. The overhead of storage reclamation is spread out over the execution time of the interpreter.

A major limitation of reference counting is the difficulty in reclaiming cyclic structures. The design of Omega prevents this problem. Only "pure" lists are used and rplacx operations are not defined.

Reference counting places a computational burden--incrementing reference counts--at a sensitive point: memory allocation. The execution penalties associated with reference counting are examined in the next chapter.

F. GARBAGE COLLECTION

Garbage collection was not selected as a storage management strategy because of the complexity of implementation. The **malloc** and **free** routines offer a predefined storage allocation system, while a garbage collection system requires an explicit design of these components.

The development of a garbage collector would be an interesting extension to the current design. Some of the considerations for a mark-and-sweep garbage collector are:

- Structure access is required for the mark phase. This phase needs to access the following interpreter structures:
    1. The object table.
    2. The active rule table.
    3. Rules under evaluation by the console command processor.
    4. Rules under evaluation by the file command processor.
    5. Intermediate results generated during rule evaluation.

Accessing intermediate results is complicated by the present recursive implementation. A possible solution is to maintain a stack specifically for referencing these structures during a mark phase.

- Cells require a mark bit. The current cell structure provides an 8 bit tag. Bits 0 through 5 are used for the tag value, bit 7 is used as a flag on certain structures. Bit 6 remains available for marking purposes.

- Complete storage access is required for the sweep phase. This immediate access implies that the memory allocator must be managed by the interpreter. To maintain a reasonable virtual memory image, this allocator must obtain and release memory on page boundaries, using compaction whenever possible.

G. REDUCING CELL STORAGE

The current 12 byte requirement for cell storage is large. Since the cell structure is based on the LISP model, numerous LISP techniques may be used to reduce this requirement.

LISP structures have fewer distinct cell types. Instead of encoding tag information directly, storage for cells of different types are often allocated from noncontiguous, separate sections of memory [Ref. 29]. In this way, the address range of a pointer provides the necessary type information.

List linearization techniques are another way to reduce storage requirements. These techniques attempt to maintain cells, normally linked via their tail pointers, in contiguous storage. This allows a reduced tail field size, with a

field containing a cell offset to the next cell in the structure instead of a full pointer. An escape mechanism allows full pointer access where necessary [Ref. 30: p. 266].

The above techniques are mentioned as potential improvements in the current allocation scheme. These techniques, like garbage collection, require more explicit control of storage allocation than is offered by the current design.

A potential storage savings can be obtained with the current tagged cell structure by embedding the tag in the tail field. This is a simple modification that requires masking the tail field value to obtain the tag or tail pointer. If a reference counting field is not used (assuming garbage collection instead) this technique reduces the current cell requirement to 8 bytes, a 33 percent reduction. Using this encoding scheme, the tail pointer is restricted to a 24 bit range.

VII. PERFORMANCE EVALUATION

A. METHODOLOGY

A progressive series of programs were developed to test features as they were implemented. These programs assisted in determining the interpreter's reliability and execution characteristics. Execution profiles and comparative benchmarks were used to evaluate behavior and performance.

In this implementation, performance was subordinate to a clear, workable design. Performance optimization efforts were started only after the design and implementation of a series of features were complete, with all test programs successfully executing.

1. Execution Profiling

The gprof call graph execution profiler [Ref. 26] was an important tool for evaluating weak points in the performance of the interpreter. Execution profiles pointed out some immediate inefficiencies in the implementation that could be easily remedied.

A simple example is the tag function. Initially, a function was used to extract the tag value from a cell. The rationale behind this implementation was information hiding: the details of the cell structure were accessible to only a few handling routines.

Execution profiles on pattern-matching showed this implementation to be costly: the interpreter spent over 10 percent of its execution time extracting tags. To solve this problem, the function was rewritten as a C macro [Ref. 22: p. 86]. The VAX instructions generated by the alternative implementations are shown in Figure 7.1. The

macro implementation results in a savings of 4 instructions--a substantial improvement given the high

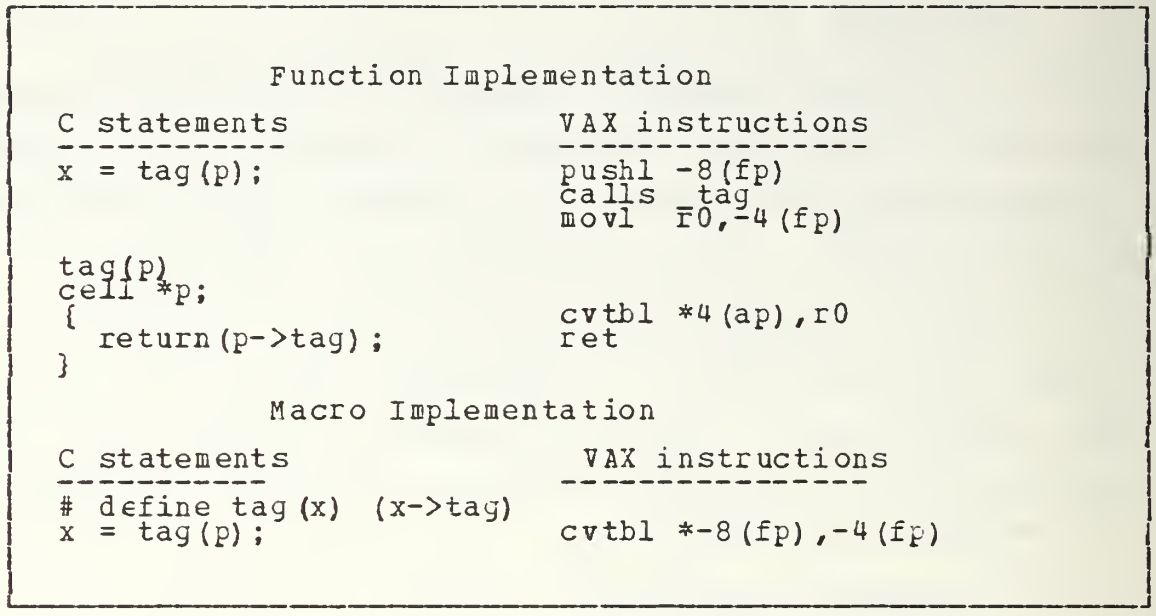

Figure 7.1 Code generation for TAG function.

frequency of tag extraction during interpretation.

Replacing procedural implementations with macros is not a panacea. Macro implementation has at least two disadvantages:

- Debugging is more difficult. Macros are textually expanded by a preprocessor before compilation. The errors that occur are unusual, do not correspond well with source code, and may produce unexpected effects.
- Profiling information is lost. An execution profile pointed out the expense of the tag function. Once coded as a macro, the cost of this code sequence is absorbed in the routines where the macro is expanded.

2. Benchmarking

In this chapter we present a collection of simple benchmark programs. The performance of the Omega interpreter is compared to interpreted Franz LISP, compiled Franz LISP, and the C-Prolog interpreter. The C-Prolog interpreter is a VAX Prolog implementation descended from the DECsystem-10/20 Prolog system [Ref. 31 and 32]. These systems are all written in C.

These benchmarks are not intended as an evaluation of C-Prolog or Franz LISP, and no effort has been made to write efficient Prolog or LISP. Because these systems are well-engineered and efficient in what they do, we present these benchmarks as an indication of the current progress of our implementation. The source code for the benchmarks is included in Appendix D.

Timing information was obtained through calls to the **date** function of the UNIX command shell [Ref. 26]. The following Omega rule demonstrates this technique:

```
if *qTest(a) -> {
     System{"date"};
     Qsort{IotaR[1, 150]};
     System{"date"};
     }.
```

The **date** function returns the current system time to the nearest second. The test systems all possessed a function similar to the **System** procedure shown above, and the overhead of executing such a system call should be consistent between interpreters. The timing granularity of one second required establishing benchmarks of sufficient duration to provide meaningful comparisons.

3. Omega Statistics

Besides benchmark and profiling measurements, additional information was collected to begin a characterization of Omega program behavior. This information included:

- Data type frequencies.
- Hash collisions in the object table.
- Relation characteristics: relation cardinality and tuple cardinality.

The last two areas are dynamic characteristics which change as rules fire and alter the database. These measurements were taken using a sampling technique: object table measurements were made immediately after each rule evaluation.

B. TEST RESULTS

Benchmark programs were tested using two different versions of the interpreter. The versions were:

- The standard interpreter without reference counting.
- A version compiled with the gprof profile option and containing object table and cell measurement routines.

The overhead of measurement and profiling necessitated the separate compilations.

The execution profiles produced by gprof are extensive. These are summarized and included in profile summary tables, with the profiling information for the five most expensive (in execution time) routines shown. The percentages given in these profile summaries are taken directly from the profile reports, and reflect the large overhead of the profiler. The profiling routines typically consumed about 50 percent of the total execution time. Thus, if an

execution percentage for a routine is shown to be 5 percent, its relative impact is approximately 10 percent in unprofiled execution.

1. A Pattern-Matching Test

This benchmark asserts a common tuple in two relations, followed by 1000 disjoint assertions to each. A pattern-match search finds the common tuple. The search requires more than 2 x 10^6 pattern-match tests, a worst-case performance.

The Prolog version is similar, with assertions made to the Prolog rule base. Prolog searches the rule base from top to bottom. The common clauses are asserted and subsequent clauses placed before these using the asserta predicate [Ref. 21: p. 105].

We include a LISP implementation of a nested loops search, although this is a simplification of the pattern-matching process of Omega and Prolog. Only a compiled LISP version was tested because an interpreted version is at an unfair disadvantage when competing with the direct implementations of this process.

The timing results for this benchmark are shown in Table II. A summary of the Omega execution profile is shown in Table III, and type information is shown in Table IV. Relation characteristics are not shown for this test.

2. Factorial Functions

This benchmark exercises the applicative component of the interpreter. A recursive factorial function is executed 500 times, with each call computing Fact[15]. Larger factorials are not used because both Franz LISP and Omega experience integer overflow in their computation.

Timing results are shown in Table V, and an Omega execution profile summary in Table VI. Table VII shows the

TABLE II

Execution Times: Pattern-matching

System	Execution Time (secs)
Omega	212
Prolog	102
LISP	
Interpreted	N/A
Compiled	230

TABLE III

Execution Profile Summary: Pattern-matching

% Execution Time	No. Calls	Name	Descr
30.5	2021016	unify	pattern-matching
10.0	1010985	equ	list equality test
5.7	5035	match_s	linear list search
4.4	100400	FreeBind	reset frame bindings
0.9	70019	eval	evaluation function

Omega type distribution for this benchmark, and Table VIII shows the relation characteristics.

3. A Prime Number Sieve

The third benchmark is a prime number generation program. The benchmarks use a sieve algorithm to remove prime number multiples from a list of numbers. This is an interesting benchmark for Omega in that the sieve is driven by rules, but the major computation--removing multiples--is

TABLE IV

Data Type Frequencies: Pattern-matching

Type	Frequency	% of Total
Lists		
Op List	2053	11
Data List	8815	49
Atoms		
Integer	2028	11
String	1141	6
Boolean	3004	17
Object	235	1
Variable	747	4

TABLE V

Execution Times: Factorial

System	Execution Time (secs)
Omega	23
Prolog	99
LISP	
Interpreted	12
Compiled	4

performed by the applicative component. The Prolog version is based on an example given in [Ref. 21: p. 157].

The timing results for this benchmark are shown in Table IX. These times are based on a sieve list of 350 integers. A summary of the Omega execution profile is given in Table X, data type distributions are given in Table XI, and relation characteristics in Table XII.

TABLE VI

Execution Profile Summary: Factorial

% Execution Time	No. Calls	Name	Descr
14.8	155457	eval	evaluation function
3.7	32059	malloc	memory allocation
3.2	24001	binop	binary operation
2.1	30346	newcell	create a new cell
1.9	9097	tupl	eval tuple/args

TABLE VII

Data Type Frequencies: Factorial

Type	Frequency	% of Total
Lists		
Op List	2001	7
Data List	9777	32
Atoms		
Integer	15530	51
String	1607	5
Boolean	502	2
Object	226	1
Variable	720	2

4. Quicksort

This benchmark exercises the Omega procedure call and pattern-matching mechanisms. The Quicksort splits lists, recursively sorts the sublists, and combines the results. The Prolog version is taken from the example given in [Ref. 21: p. 147].

TABLE VIII

Omega Relation Characteristics: Factorial

Sample Frequency: 2078

Characteristic	Mean	Std Dev	Mode
Relation cardinality	1.0	0.0	1.0
Tuple cardinality	2.0	0.1	2.0

Object Table Collisions:

Collision List Length	Frequency
1	43126

TABLE IX

Execution Times: The Sieve

System	Execution Time (secs)
Omega	19
Prolog	11
LISP	
Interpreted	9
Compiled	3

The timing results are shown in Table XIII. These times are based on executing an ascending sort on a list of 150 integers initially arranged in descending order (an $O(n^2)$ undertaking for Quicksort). An execution profile summary is given in Table XIV, and Table XV contains the data type frequencies. The relation characteristics of the benchmark are shown in Table XVI.

TABLE X

Execution Profile Summary: The Sieve

% Execution Time	No. Calls	Name	Descr
13.7	118635	eval	evaluation function
4.2	16462	tupl	eval tuple/args
4.0	29425	malloc	memory allocation
2.9	28169	newcell	create new cell
2.7	12348	fnapl	apply fn to args

TABLE XI

Data Type Frequencies: The Sieve

Type	Frequency	% of Total
Lists		
Op List	1990	7
Data List	20372	72
Atoms		
Integer	3266	12
String	1146	4
Boolean	423	2
Object	232	1
Variable	761	3

5. A Simulation Program

This example is a Monte Carlo simulation of a three node message switching network. It is a more complex program than the preceding benchmarks, and the interpreter exhibits a wider range of activities. The source code for the simulation rules is listed in Appendix E.

TABLE XII

Omega Relation Characteristics: The Sieve

Sample Frequency: 149

Characteristic	Mean	Std Dev	Mode
Relation cardinality	1.0	0.0	1.0
Tuple cardinality	2.6	0.5	3.0

Object Table Collisions: Collision List Length	Frequency
1	2694

TABLE XIII

Execution Times: Quicksort

System	Execution Time (secs)
Omega	142
Prolog	27
LISP	
Interpreted	81
Compiled	40

Comparative benchmarks were not written for this problem, and timing comparisons are not shown. The execution profile for the simulation is given in Table XVII, data type frequencies are shown in Table XVIII, and relation statistics are shown in Table XIX.

TABLE XIV

Execution Profile Summary: Quicksort

% Execution Time	No. Calls	Name	Descr
13.7	717396	eval	evaluation function
5.1	216385	unify	pattern-matching
2.7	151009	malloc	memory allocation
2.6	91975	tupl	eval tuple/args
2.0	143777	newcell	create new cell

TABLE XV

Data Type Frequencies: Quicksort

Type	Frequency	% of Total
Lists		
Op List	1753	1
Data List	121627	85
Atoms		
Integer	170	0
String	6820	5
Boolean	11777	8
Object	533	0
Variable	808	1

C. IMPACT OF REFERENCE COUNTING

The timing information presented previously was taken without reference counting. A separate version of the interpreter was compiled with the reference counting routines included, and separate measurements taken. The impact of reference counting on execution speed is shown in Table XX. A summary of the Quicksort benchmark, with reference counting implemented, is shown in Table XXI.

TABLE XVI

Omega Relation Characteristics: Quicksort

Sample Frequency: 35704

Characteristic	Mean	Std Dev	Mode
Relation cardinality	1.0	0.0	1.0
Tuple cardinality	4.9	0.7	5.0

Object Table Collisions:

Collision List Length	Frequency
1	785578
2	264

TABLE XVII

Execution Profile Summary: Simulation

% Execution Time	No. Calls	Name	Descr
9.4	211360	eval	evaluation function
3.3	63126	unify	pattern-matching
2.9	63782	malloc	memory allocation
2.1	1221	write	system i/o
1.8	35220	lookup	hash table lookup

D. DISCUSSION OF RESULTS

1. Performance Bottlenecks

The measurements presented in the preceding sections provide some insight into the effectiveness of the present

TABLE XVIII

Data Type Frequencies: Simulation

Type	Frequency	% of Total
Lists		
Op List	6125	11
Data List	26053	47
Atoms		
Integer	2147	4
String	6340	11
Boolean	10124	18
Object	2390	4
Variable	2290	4

TABLE XIX

Omega Relation Characteristics: Simulation

Sample Frequency: 9972

Characteristic	Mean	Std Dev	Mode
Relation cardinality	5.7	6.3	1.0
Tuple cardinality	2.3	0.7	2.0

Object Table Collisions: Collision List Length	Frequency
1	404828
2	180974
3	26152
4	4552
5	2761
6	11

implementation. The execution speeds for the Omega benchmarks are consistently slower than C-Prolog and Franz LISP.

TABLE XX

Reference Counting and Execution Times

Benchmark	w/o Ref Counts	w Ref Counts	%Incr
Pattern match	212	223	5
Factorial	23	30	30
Sieve	19	25	32
Quicksort	142	181	27

TABLE XXI

Profile of Quicksort with Reference Counting

% Execution Time	No. Calls	Name	Descr
9.9	717396	eval	evaluation function
4.0	216385	unify	pattern-matching
3.7	421556	DecrRef	decrement ref count
3.3	541624	IncrRef	increment ref count
2.4	151009	malloc	memory allocation

This performance is shown in both applicative expression and rule evaluation.

The central evaluation function, eval, consumes the majority of execution time. This is not surprising given the present recursive implementation: **eval** is called directly or indirectly in most operations. Embedded in eval are accesses of the binding stack for atom evaluation in tuples and argument lists.

The high frequency of calls to **eval** suggest its design as a potential point for optimization. This

optimization may include a reduction of unnecessary recursive calls to **eval**. When evaluating interior nodes of a rule tree, the sons of a node are always passed to **eval**, which will decode the tag and call the appropriate subordinate routine. If the required subordinate routine is known at the parent node, the intermediate call to eval may be omitted.

Another alternative is to replace the recursive **eval** with an iterative version. This requires the management of an explicit operand stack, and involves a major redesign effort. The management of an operand stack would, however, solve an implementation problem for garbage collection discussed in the previous chapter.

The pattern-matching routine becomes significant in extended rule processing, as shown by the Quicksort and network simulation tests. We note similarities between **eval** and the pattern-matching routine **unify**:

- Both routines are heavily exercised.
- Both routines access the binding stack when evaluating free variables.
- Both routines are recursive.

The recursive algorithm for pattern-matching is simple and elegant. An iterative version would be more complex, but may provide a performance gain.

A final area for performance improvement is storage management. The storage allocation routine, **malloc**, and routines that call it, such as **newcell**, consistently rank high in the execution profiles. Our implementation of reference counting for storage reclamation proved to be expensive, with a 30 percent increase in execution time for extended tests. These results make garbage collection appear to be a desirable alternative. Hardware support for reference counting could also provide a solution.

2. Relation Statistics

The statistical information gathered on relations provides some evidence to support previous conjectures:

- Small relations are commonplace. The mode for relation cardinality was 1.0 in every test. These statistics will vary depending on the application, and generalizations can't yet be made based on the limited tests conducted.
- Object identifiers hash well. The object table collision results indicate an even distribution of hash values. Collisions in the object table slow down relation list lookup, an important part of the synchronous procedure call. Only in the simulation test did hash table lookups begin to become significant in the execution profiles. This coincides with the increased number of objects generated and an increased collision frequency in the object table.

VIII. OBSERVATIONS, RECOMMENDATIONS, AND CONCLUSIONS

A. OBSERVATIONS ON OMEGA

1. Programming Experience

Our experience with Omega programming is reflected in the rules listed in Appendices C-E. We believe these examples demonstrate a variety of applications which have simple solutions in Omega rules.

An important body of rules are the system utilities listed in Appendix C. Included are relation copying utilities, an extension to the system pretty printer, and a help facility. The last application shows a simple use of the **System** procedure call to list help files at the user's terminal. This technique could be extended to use the Omega interpreter as a rule-based driver for the UNIX command shell.

The longest and most significant application is the simulation model listed in Appendix E and profiled in the previous chapter. We include this example as an event-driven, state-transition problem which is readily expressed as rules of the form:

```
if Clock(t1), *Event(t1, e) ->
     ProcessEvent(e).
```

2. Omega and Prolog

The benchmarking examples of the previous chapter presented rules in Prolog and Omega that are similar in form. Both these languages use pattern-directed invocation for rule selection, and both languages are intended for

general programming applications. Despite similarities, these languages have fundamental differences.

Omega uses forward inference, Prolog uses backward inference; Omega programmers and Prolog programmers think in different directions. To design an Omega rule, one uses the train of thought: "Given the current state of the system, generate the next state." A Prolog rule is designed with the thought: "To prove this goal, it is necessary to prove these subgoals." Prolog relies on a theorem-proving approach, Omega on a data-driven approach.

These opposite control strategies are reflected in different implementation techniques:

- Prolog recursively evaluates its rules from a goal stack. This technique often allows intermediate storage allocation from stack structures. This method of allocation and reclamation is simpler than the heap allocation used by Omega and LISP, and results in a faster cons operation [Ref. 32: p. 114]. This performance is reflected in the Quicksort benchmark of the previous chapter.

- Theorem proving requires backtracking. Prolog selects rules from its rule base to prove subgoals. If multiple rules for a subgoal are present, they will be selected and tested until the subgoal is proven or all possible rules fail. Backtracking between rules requires a more general pattern-matching technique in Prolog than Omega. In Prolog, variables may be bound to variables. In Omega, a rule fires or it doesn't--there is no requirement for backtracking between rules, and variables are bound only to objects and values in the database.

Programming problems may be solved by either forward or backward inference. To illustrate this point, we use the

missionaries and cannibals problem [Ref. 33: p. 51], a simple state-space search example. The Omega and Prolog rules for this problem are listed in Appendix D.

In the missionaries and cannibals problem, a simplified description of the Omega rules is:

```
if *State(x, path), GoalState[x] ->
     Displayn{path}
else if *State(x,path), IsIegalState[x],
     ¬member[x, path] ->
     GenerateNewStates{x,[x:path]}
else if *State(x,path) ->.
```

Given a starting state, the Omega rules will generate all possible new states that may be reached from that state. This process continues until all combinations of legal states have been tested. No backtracking is required in these rules--successful states continue to fire, and unsuccessful states are removed from the computation. We maintain a list of previous states in the variable path to prevent cycles.

A simplified version of the Prolog rules for this problem is:

```
goalState(X,Path) :-
     finalState(X),
     print(Path).
goalState(X,Path) :-
     not member(X,Path),
     legalState(X),
     possibleNextState(X,Y),
     goalState(Y,[X|Path]).
```

In the Prolog version, the predicate possibleNextState will bind the variable Y to a new state that can be reached from X. In its attempt tc "prove" the starting goal state, all

possible state combinations will be generated by backtracking on possibleNextState. We observe Prolog exploring new states through backtracking where Omega relies on the generation of new states in the database.

Certain classes of problems lend themselves well to the natural recursion inherent in Prolog. The Quicksort rules of Appendix D are a model of brevity and clarity. We suggest that event-driven or data-driven applications, such as the simulation example of Appendix E, are better described through Omega. Omega was developed as a high-level language for programming environment description and implementation. This family of applications are represented more naturally through forward inference descriptions.

3. The Production Rule as a Programming Paradigm

Both Omega and Prolog use the production rule as the programming paradigm. How easy is it to program with production rules? We consider this to be an application-dependent quality. A problem can be effectively described with production rules if the following characteristics apply:

- The problem can be decomposed into a set of small, cause/effect subproblems. Each subproblem is described by a single rule or small set of rules.
- The subproblems are independent, and require minimal communication between rules.

The independence of rules allows the programmer to add or remove rules without concern for the impact of these changes on other rules in the active rule list. In our current design, the rule denctation is the unit of rule organization. Thus, a goal for a manageable rule structure is independence of rule denotation sets.

A significant limitation of most production rule systems is a lack of meaningful semantic composition, the inability to compose a complex action from a collection of previously defined simpler actions. Rosenchein writes:

> [In production systems] tests and transformations are sophisticated and are designed to implement constructs found in various applications. However, there is generally no way to symbolize composition of operations in a transparent way. Complicated tests and actions have to be simulated by groups of rules whose coordination is not symbolized in the program or graced with a mnemonic name. The more complicated the tests and actions, the more severe the coordination problems. This is typical of programs written at one level of abstraction, no matter how sophisticated the primitive operations. [Ref. 34: p. 535]

Omega provides some potential solutions to this problem. The object-oriented approach of the language allows the partitioning of related data and rules through directories and classes. This organization of the name space provides the first step in a hierarchical composition of rule activity.

The second step is the procedure call mechanism. This mechanism serves as an invocation trigger for a collection of rules, with a method of integrating the outcome from their actions into more complex expressions. The utility of this mechanism is indicated by its widespread use in the examples presented in this thesis.

Although our programming examples are dependent on the procedure call, we note potential problems with the mechanism:

- The actions associated with a procedure call may not be obvious. The procedure call asserts a tuple to a given relation, and extracts the response from its mailbox. Multiple rule denotations may use the procedure's relation, and fire as a result of the procedure assertion; the possibilities for subtle side effects are

significant. The final effect of a procedure call can only be determined from an analysis of all rules associated with the procedure's relation name (as defined in its directory).

- The procedure mechanism does not enforce parameter checks. The tuple asserted in a procedure call is analogous to the actual parameters of a conventional procedure call. In Omega, there is no parameter counting or type checking. Consider the following rule:

  ```
  if *R(a, x) -> displayn{x}.
  ```

  If the user mistakenly enters "R(100)," the assertion will be made but the rule will not fire because of a pattern-matching failure. The procedure call "R{100, 200}" will fail for the same reason. In both of these situations, no error indication will be given.

There are programming techniques that correct the last problem. If we code the rule as:

```
if *R(a:l) -> . . .
```

the head/tail list specification will match against any tuple. The rule designer may then code explicit type and parameter count checks with an appropriate response to errors. We use this technique in the utility rules contained in Appendix C.

A declarative mechanism may also be used to specify the expected tuple size for a given relation. Any deviations would trigger an error response.

B. RECOMMENDED AREAS FOR ADDITIONAL STUDY

1. Extensions to the Language

Our programming and implementation experience with Omega have suggested three additional extensions to the language: (1) a syntactic distinction for free variables, (2) a universal quantifier, and (3) named rules.

In our current implementation, the distinction between free and bound variables is made when rules are activated: if defined in the class/directory structure, the variable is bound; if not defined, it is considered free.

This strategy is a potential source of error. Suppose we wish to define a constant, and use the following definition:

```
Define{Root, "a", 100}.
```

The selection of the variable name a will conflict with the majority of our rule denotations, where this variable is consistently used to represent a mailbox relation. The activation and test of the follcwing rule:

```
if *T(a, x) -> a(2 * x).
```

will result in a type clash error. These errors are subtle, and require the programmer tc remember which names are previously defined in a given environment. To solve this problem, free variables should be syntactically distinguished. Thus, the preceding rule may be written:

```
if *T(&a, &x) -> &a(2 * &x).
```

Previous definitions cannot adversely affect this rule. C-Prolog uses a similar convention: free variable names begin with upper case letters, bound variable names begin with lower case letters.

In Chapter 2, we emphasized the point that a relation inquiry is existentially quantified. Consider the following rule:

```
if CopyRel(r1, r2), r1(x), ¬r2(x) ->
     r2(x)
else if *CopyRel(r1, r2) ->.
```

This rule implements a relation copying utility. The programmer's intent for this rule is that all tuples in r1 should be asserted to relation r2. Existential quantification will select a single tuple on each firing cycle for the rule. Note that the absence test on relation r2 is required to ensure termination.

A possible alternative is to provide universal quantification for tuple selection. With this mechanism, the copy rule could be written:

```
if *CopyRel(r1, r2), $r1(x) -> r2(x).
```

The "$" symbol is used to represent universal quantification. The action of the quantifier is "for all tuples x in relation r1, assert x to relation r2." A universal quantifier offers the following advantages:

- The programmer's intentions are more clearly expressed. There is a similarity between universal quantification and the mapcar function of LISP.
- Performance may be enhanced. A universal quantifier allows optimization by completing the actions of the rule in one rule cycle. The costs of multiple rule selection and testing, shown in the profiles of the unify procedure in the preceding chapter, may be high.

It is recognized that the existential quantification of the present design is sufficient to accomplish the

intended function of the CopyRel rules and similar applications. The advantages of universal quantification must be weighed against the added complexity to the language.

It would be useful to be able to reference rules by name. The rule forms the basic computational unit in the language, yet may not be referenced explicitly; the only named reference for a rule is its source rule denotation. If the denotation is large, its name is not a selective description. If a single rule is to be manipulated, perhaps by a structure editor, the entire denotation must be accessed. A similar problem is experienced with activating and deactivating rule denotations.

2. Extensions to the Present Interpreter Design

Our present implementation includes the majority of Omega language features described in [Ref. 14]. Several possible extensions to the present implementation are of interest.

The class mechanism described in Chapters II and IV is not currently implemented. The present system provides a single directory, root, frequently referenced in our examples. The class and directory structures, with rules indexed on relation objects, provide the inheritance mechanism for the language. The implementation of classes as binding lookup paths will allow a more thorough exploration of the object-oriented nature of Omega than has been provided in this work.

Both the LISP prototype and the current prototype use a simple linked-list relation structure. There are two alternate representations that are of immediate interest:

- Hashed indexing of relation lists. Relations lists are currently selected via hashed access of the object table. A performance-enhancing extension to this

technique is to provide a hashed index structure for tuples within a relation, possibly using a user-specified key or arbitrarily using the first tuple member as a key. An indexed relation structure would have little impact on most of the benchmarks of the preceding chapter (except the pattern-matching test) because of the small relation sizes. A substantial performance improvement for inquiries on larger relations may be realized.

- A relational DBMS implementation of relations. The current design can be extended to include a query and response translation interface with a concurrently executing DBMS. The interface could be organized around object identifier recognition, as system objects are currently handled. The tag field of DBMS-supported objects could be assigned a unique value to route these objects to the DBMS interface instead of the normal relation management routines.

Control strategies for rule selection and test remain an open subject in this work. Our two-level triggering strategy, discussed in Chapter V, was selected as a compromise implementation that provides a reasonable level of rule selection precision under a programmer's control. Full indexing of rules--triggering on all the relations in the rule antecedent--was not attempted. The performance of full indexing compared to two-level triggering is a potential point for additional study.

The control strategy of Omega, like Prolog, is "hard-wired" into the interpreter design. The designs of several other rule-based systems have taken a more flexible approach: meta-rules dictate control strategies [Ref. 35]. The idea of integrating rule-based control strategies into Omega would be an interesting extension to the language and its interpretation.

The space considerations of Chapter VI, as well as the execution statistics of Chapter VII, indicate that a garbage collection scheme may preferable to reference counting. The implementation of an efficient compacting garbage collector can provide significant performance improvements.

A useful extension to our implementation would be a flexible debugger. We currently use a trace facility that provides a display of rule execution. While this facility is useful, the information provided is not specific enough. The following debugging features would be helpful:

- Specification of trace and break points by relation name. When a tuple is added or removed from a relation, the debugger may be invoked and the bindings of variables available for examination.
- Specification of trace and break points for specific rules. This facility is similar to the preceding one, but only certain rules are monitored. Note that the lack of names for individual rules makes this feature difficult to implement.
- Debugger invocation on error conditions. As in the LISP prototype, error conditions result in an error message and a return to the interpreter's top level. The implementation of a debugger would allow a more sophisticated response.

Our present method for rule creation and modification should be extended. Currently, the following steps are used to create rules:

- Command rules are entered into a file using a standard text editor. The vi procedure call is provided in Omega to allow ready access to the UNIX editor of the same name [Ref. 26].

- The file is parsed using the file command reader, invoked with the Do procedure call.
- Any rule denotations contained in the command rules are then activated.

This process requires rule files to be reloaded each time the interpreter is run. Also, the current implementation allows rule activation but not deactivation. These limitations suggest the following extensions:

- A save/restore facility. This facility would copy and restore the virtual memory image of the interpreter's data segment.
- A structure editor. There is currently no way to edit rule denotations once they are loaded into the interpreter. A structure editor would be useful, particularly when debugging.
- Rule deactivation. The ability to remove rules from active status is necessary to allow testing and modification. The use of rule names would simplify this process.

3. Parallelism in Omega

The prototypes discussed in this work have been sequential, single-threaded control implementations. An important extension to this work is the exploration of parallelism in Omega rule interpretation, with architectures tailored for concurrent rule evaluation.

Parallelism in Omega shares similarities with parallelism in Prolog. The Prolog literature divides this parallelism in two categories: AND parallelism (where multiple subgoals in a rule are evaluated concurrently), and OR

parallelism (where multiple rules for a goal are evaluated concurrently) [Ref. 36: p. 29].

In Omega, AND parallelism may be exploited in both the antecedent and consequent of a rule. In the antecedent, inquiries must be independent for concurrent evaluation. Consider the following rule:

```
if *R1(x), *R2(x), *R3(y) -> R3(x+y).
```

The first two inquiries, **R1(x)** and **R2(x)**, are mutually dependent on the variable **x**. An evaluation order for these inquiries should be made to allow the binding for the variable **x** to guide the relation search. The third inquiry, **R3(y)**, is independent of evaluation order. An AND parallel strategy should separate a rule antecedent into independent subunits that exploit concurrency where possible.

The evaluation of actions in the consequent of a rule has fewer constraints. Any actions separated by commas are potentially subject to concurrent evaluation. The actions of a sequential block are, however, restricted to a mandatory evaluation order.

The evaluation of separate rules in Omega is unordered; OR parallelism is the norm for the Omega model. A parallel implementation of rule evaluation is complicated by the shared memory access that many rules require. This shared access implies a locking mechanism at the relation level, along with a deadlock-prevention strategy.

The exploitation of parallelism in Omega offers the potential for the resolution of our present performance bottlenecks. A parallel architecture that optimizes pattern-matching and concurrent rule evaluation would yield substantial performance improvements.

C. CONCLUSIONS

This thesis has described two prototype implementations of Omega interpreters, based cn the model of a LISP-like list processing system. Through these designs, we have developed a complete LL(1)/LR(1) grammar, and implemented a table-driven parser using the YACC parser generator. Many of the design and implementation problems encountered are similar to the LISP design issues of the literature.

The unconventional component of Omega is its pattern-directed set of active rules. Our implementation processes these rules by a simple triggering method of rule indexing based on relation identifiers. Triggering, along with the processing of multiple rule queues, simulates the concurrent evaluation of rules in the Omega model.

An evaluation of our interpreter's performance indicates current execution speeds are slower than Franz LISP and C-Prolog. Possible bottlenecks include excessive recursive calls to the central evaluation function, inefficiencies in pattern-matching, and execution penalties in storage management routines. While the present design may be optimized to improve this performance, other issues are of more interest in future research. These issues include alternative representations of relations, alternative control strategies for forward inference rule systems, and the exploitation of parallelism in Omega.

Our final contribution in this work is a body of Omega programs and some statistical information on their behavior. As the Omega language matures, this experience may help to characterize potential extensicns and improvements to the language and its implementation.

APPENDIX A

LEX AND YACC SPECIFICATIONS FOR OMEGA

```
/*********************************************************
 *                                                       *
 *       LEX specification --                            *
 *       Omega lexical grammar                           *
 *                                                       *
 *********************************************************/

        /* lexical scanner grammar */


digit       [0-9]
lttr        [a-zA-Z]
whitespace  [ \t\n]
identifier  {lttr}(({lttr}|{digit}|_)*({lttr}|{digit}))*
int_con     {digit}+
delimiter   [-+!|,.;'@(){}*/%¬&:=<>_]|\[|\]
implication ->
deno        <<
end_deno    >>
ne          ¬=
le          <=
ge          >=
int_con     {digit}+
str_con     \"[^"]*\"
comment     \![^\n]*\n

%%

        /* recognizer actions for token classes */


                int c;
```

```
{identifier}    {
                        if (strequ(yytext, "if"))
                           return(IF);
                        else if (strequ(yytext, "else"))
                           return(ELSE);
                        else if (strequ(yytext, "fn"))
                           return(FUNCTION);
                        else if (strequ(yytext, "Nil"))
                           return(NIL);
                        else
                           return(IDENTIFIER);
                }

{int_con}       {
                        return(INT_CON);
                }

{str_con}       {
                        c = input();
                        if (c == '\"') {
                           unput(c);
                           --yyleng;
                           yymore();
                           }
                        else {
                           unput(c);
                           return(STR_CON);
                           }
                }


{implication}   {
                        return(IMPLICATION);
                }
```

```
{deno}          {
                        return(EEG_DENO);
                }

{end_deno}      {
                        return(END_DENO);
                }

{ne}            {
                        return(NE);
                }

{le}            {
                        return(IE);
                }

{ge}            {
                        return(GE);
                }

{delimiter}     {
                        return(yytext[0]);
                }

{comment}       ;

{whitespace}    ;

        /* ignore others */

%%

strequ(s1, s2)
char *s1, *s2;
{
        return(strcmp(s1, s2) == 0);
}
```

```
tracefun(msg)
char *msg;
{
        printf("\n***%s*** --> ",msg);
        ECHO;
        printf("\n");
}


/*********************************************************
 *                                                       *
 *         YACC Specification for Omega Parser           *
 *                                                       *
 *********************************************************/

%{

# include "tag.h"
# include "defs.h"

PTR newcell();
PTR makint();
PTR makstr();
PTR parsetree;
char  *strdeno();

%}

/* type declarations for parser stack */

%union  {
        PTR cell;
        }

%type <cell> session    statement_list
%type <cell> statement  cpd_rule    rule cause
%type <cell> conditions condition   inquiry
%type <cell> constraint transaction transactions
```

```
%type <cell> effect     assertion   denial
%type <cell> arguments  seq_block   expression
%type <cell> unop       binop       primary
%type <cell> rule_denotation
%type <cell> variable   self_ref    constant
%type <cell> fn_definition          fn_head
%type <cell> fn_application         list
%type <cell> cond_expr  cpd_expr    call

/* token declarations */

%token  IDENTIFIER
%token  IF              ELSE
%token  STR_CON         INT_CON
%token  IMPLICATION     BEG_DENO        END_DENO
%token  NE              LE              GE
%token  FUNCTION        NIL

/* operator precedance */

%left   ','
%left   expression
%left   list
%left   '&' '|'
%left   '¬'
%left   '=' '<' '>' NE LE GE
%left   '+' '-'
%left   '*' '/' '%'

%start session

%%


/* productions for Omega grammar
-------------------------------------------------------------*/
```

```
session         :       /* empty */
                                {
                                parsetree = Nil;
                                }
                |       '.'
                                {
                                parsetree = Nil;
                                return(-1);
                                }
                |       statement_list '.'
                                {
                                parsetree = $1;
                                return(-1);
                                }
                ;
statement_list  :       statement
                                {
                                $$=newcell(STA_LIST,Nil,$1);
                                }
                |       statement_list ';'  statement
                                {
                                $$=newcell(STA_LIST,$1,$3);
                                }
                ;


statement       :       cpd_rule
                                {
                                $$ = $1;
                                }
                |       fn_definition
                                {
                                $$ = $1;
```

```
                        }
        ;

cpd_rule        :       rule
                        {
                        $$=newcell(CPD_RULE,Nil,$1);
                        }
        |       cpd_rule ELSE rule
                        {
                        $$=newcell(CPD_RULE,$1,$3);
                        }
        ;

rule            :       IF cause IMPLICATION effect
                        {
                        $$=newcell(RULE,$2,$4);
                        }
        |       IF cause IMPLICATION
                        {
                        $$=newcell(RULE,$2,Nil);
                        }
        |       IMPLICATION effect
                        {
                        $$=newcell(RULE,Nil,$2);
                        }
        |       effect
                        {
                        $$=newcell(RULE,Nil,$1);
                        }

        ;

cause           :       conditions
                        {
                        $$ = $1;
```

```
                        }
                ;

conditions      :       condition
                        {
                        $$=newcell(CONDS,Nil,$1);
                        }
                |       constraint
                        {
                        $$=newcell(CONDS,Nil,$1);
                        }
                |       conditions ',' condition
                        {
                        $$=newcell(CONDS, $1, $3);
                        }
                |       conditions ',' constraint
                        {
                        $$=newcell(CONDS, $1, $3);
                        }
                ;

condition       :       '*' inquiry
                        {
                        $$=newcell(CANC,$2,Nil);
                        }
                |       '¬' inquiry
                        {
                        $$=newcell(ABST,$2,Nil);
                        }
                |       inquiry
                        {
                        $$=newcell(PRES,$1,Nil);
                        }
                ;
```

```
inquiry         :       primary '(' arguments ')'
                                {
                                $$=newcell(INQY,$1,$3);
                                }
                ;

constraint      :       expression
                                {
                                $$=newcell(CONSTR,$1,Nil);
                                }
                ;
effect          :       transactions
                                {
                                $$ = $1;
                                }
                ;
transactions    :       transaction
                                {
                                $$=newcell(TRANS,Nil,$1);
                                }
                |       transactions ',' transaction
                                {
                                $$=newcell(TRANS,$1,$3);
                                }
                ;
transaction     :       assertion
                                {
                                $$ = $1;
                                }
                |       denial
```

```
                        {
                        $$ = $1;
                        }
                |       expression
                        {
                        $$ = $1;
                        }
                |       seq_block
                        {
                        $$ = $1;
                        }
                ;

assertion       :       primary '(' arguments ')'
                        {
                        $$=newcell(ASSERT,$1,$3);
                        }
                ;

denial          :       '¬' primary '(' arguments ')'
                        {
                        $$=newcell(DENY,$2,$4);
                        }
                ;

fn_definition   :       FUNCTION fn_head ':' cpd_expr
                        {
                        $$=newcell(FN,$2,$4);
                        }
                ;

fn_head         :       '[' arguments ']'
                        {
                        $$=newcell(FNDCL,Nil,$2);
                        }
                |       primary '[' arguments ']'
                        {
```

```
                        $$=newcell(FNDCL,$1,$3);
                        }
        ;

arguments       :       /* empty */
                        {
                        $$ = Nil;
                        }
        |               list
                        {
                        $$ = $1;
                        }
        |               expression ':' expression
                        {
                        $$=newcell(CONS,$1,$3);
                        }
        ;

list            :       expression
                        {
                        $$=newcell(LIST,Nil,$1);
                        }
        |               list ',' expression
                        {
                        $$=newcell(LIST,$1,$3);
                        }
        ;


seq_block       :       '{'  statement_list '}'
                        {
                        $$=newcell(SEQ_BLK,$2,Nil);
                        }
```

```
                |       '{'  statement_list ';' '}'
                                {
                                $$=newcell(SEQ_BLK,$2,Nil);
                                }
                ;
expression      :       primary
                                {
                                $$ = $1;
                                }
                |       constant
                                {
                                $$ = $1;
                                }
                |       call
                                {
                                $$ = $1;
                                }
                |       fn_application
                                {
                                $$ = $1;
                                }
                |       rule_denotation
                                {
                                $$ = $1;
                                }
                |       unop
                                {
                                $$ = $1;
                                }
                |       binop
                                {
                                $$ = $1;
                                }
                |       '(' cpd_expr ')'
```

```
                        {
                        $$ = $2;
                        }
                |       '[' list ']'
                        {
                        $$ = $2;
                        }
                |       '[' expression ':' expression ']'
                        {
                        $$=newcell(CONS,$2,$4);
                        }
                |       '[' ']'
                        {
                        $$ = Nil;
                        }
                ;

fn_application  :        primary '[' arguments ']'
                        {
                        $$=newcell(APL,$1,$3);
                        }
                ;
call            :       primary '{' arguments '}'
                        {
                        $$=newcell(CALL,$1,$3);
                        }
                ;
rule_denotation :        BEG_DENO statement_list END_DENO
                        {
                        $$=newcell(DENO,$2,Nil);
                        }
                |       BEG_DENC statement_list ';' END_DENO
                        {
```

```
                        $$=newcell(DENO,$2,Nil);
                        }
                ;

unop            :       '+' expression  %prec '*'
                        {
                        $$ = $2;
                        }
                |       '-' expression  %prec '*'
                        {
                        $$=newcell(UNMINUS,$2,Nil);
                        }
                |       '¬' expression
                        {
                        $$=newcell(NOT,$2,Nil);
                        }
                ;

binop           :       expression '+' expression
                        {
                        $$=newcell(PLUS,$1,$3);
                        }
                |       expression '-' expression
                        {
                        $$=newcell(MINUS,$1,$3);
                        }
                |       expression '*' expression
                        {
                        $$=newcell(MULT,$1,$3);
                        }
                |       expression '/' expression
                        {
                        $$=newcell(DIV,$1,$3);
                        }
                |       expression '%' expression
                        {
```

```
                $$=newcell(MOD,$1,$3);
                }
        |       expression '=' expression
                {
                $$=newcell(EQU,$1,$3);
                }
        |       expression '<' expression
                {
                $$=newcell(LT,$1,$3);
                }
        |       expression '>' expression
                {
                $$=newcell(GT,$1,$3);
                }
        |       expression NE  expression
                {
                $$=newcell(NEQ,$1,$3);
                }
        |       expression LE  expression
                {
                $$=newcell(LEQ,$1,$3);
                }
        |       expression GE  expression
                {
                $$=newcell(GEQ,$1,$3);
                }
        |       expression '&' expression
                {
                $$=newcell(AND,$1,$3);
                }
        |       expression '|' expression
                {
                $$=newcell(OR,$1,$3);
                }
```

```
                ;

primary         :       variable
                                {
                                $$ = $1;
                                }
                |       self_ref
                                {
                                $$ = $1;
                                }
                ;

self_ref        :       '@' variable
                                {
                                $$=newcell(SELF_REF,$2,Nil);
                                }
                ;


variable        :       IDENTIFIER
                             {
                             parsetree = makstr(yytext);
                             $$=newcell(VAR,Nil,parsetree);
                             }
                ;

constant        :       STR_CON
                             {
                             $$ = makstr(strdeno(yytext));
                             }
                |       INT_CON
                                {
                                $$ = makint(atoi(yytext));
                                }
                |       NIL
                                {
                                $$ = Nil;
```

```
                        }
        ;

cpd_expr        :       cond_expr
                        {
                        $$=newcell(CPD_RULE,Nil,$1);
                        }
                |       cpd_expr ELSE cond_expr
                        {
                        $$=newcell(CPD_RULE,$1,$3);
                        }
        ;
cond_expr       :       expression
                        {
                        $$=newcell(RULE,Nil,$1);
                        }
                |       IF constraint IMPLICATION expression
                        {
                        $$=newcell(RULE,$2,$4);
                        }
        ;
```

```
%%

/* user defined functions */

# include "lex.yy.c"


/*      string denotation routine
Lexical scanner returns string constants
with " delimiters included.
Denotation routine simply removes the extra "s.
-----------------------------------------------------------*/

char *strdeno(s2)
char *s2;
{
        char *s1;

        s1 = s2 + 1;
        * (s1 + (strlen(s1)-1)) = '\0';
        return(s1);
}


/*      error handler
Prints error message and line number where error occurred.
No attempt made to recover from errors.
-----------------------------------------------------------*/
yyerror(msg)
char *msg;
{
        printf("%s : ", msg);
        printf("line %d\n", yylineno);
}
```

APPENDIX B
BUILT-IN FUNCTIONS AND PROCEDURES

I. Built-in Functions

A. Infix operators --

op	type
+	str x str -> str (concatenation)
+	int x int -> int
-	"
*	"
%	" (modulus)
/	"
<	int x int -> Bool
>	"
<=	"
>=	"
=	any x any -> Bool
¬=	any x any -> Bool
&	Bool x Bcol -> Bool
|	Bool x Bcol -> Bool

Unary operators :

-	Int -> Int
¬	Bool -> Bool

B. System defined functions :

IsInt[item] -- Returns TRUE if item is an integer.

IsStr[item] -- Returns TRUE if item

is a string.

IsList[item] -- Returns TRUE if item is a list.

IsObj[item] -- Returns TRUE if item is an object.

IsRel[item] -- Returns TRUE if item is a relation object.

int_str[int] -- Integer to string conversion.

first[list] -- CAR. Returns the first member of a list.

rest[list] -- CDR. Returns all members of a list after the first member.

cons[x, l] -- Returns a new list with x as its first member and l as the rest of the list. The notation [x:l] does the same thing.

objval[object] -- Returns the value bound to an object.

The following objects have bound values :
relations
functions
rule denotations
directories

Tag[form] -- Returns a string representing the tag of form.

II. Built-in Procedures

A. General usage:

do{"filename"}	-- Executes rules from file.
activate{name}	-- Activate rule denotation.
decode{form}	-- Post order dump of nodes.
display{form}	-- Pretty printer.
displayn{form}	-- Pretty printer. Ends with a newline.
define{dir,"name",def}	-- Make a directory entry.
trace{}	-- Toggle the trace function.
exit{}	-- End session. Same as Cntl-D.
newobj{}	-- Generate a new object.
newrel{}	-- Generate a new relation.
purge{relation_name}	-- Empty a relation.

B. UNIX system hooks:

vi{"file"}	-- Invoke the VI text editor.
system{"command"}	-- Pass a command to the UNIX shell.
shell{}	-- Spawn a temporary UNIX shell.

APPENDIX C
UTILITY FUNCTIONS AND RULES

```
!!!
```

```
        if l1 = Nil | l2 = Nil -> Nil
        else cons[[first[l1], first[l2]],
                  pairlist[rest[l1], rest[l2]]].

fn append[l1, l2] :
        if l1 = Nil -> l2
        else if l2 = Nil -> l1
        else cons[first[l1], append[rest[l1], l2]].

fn iota[n1, n2] :
        if n1 > n2 -> Nil
        else cons[n1, iota[n1+1, n2]].

fn length[l] :
        if l=Nil -> 0
        else 1 + length[rest[l]].

fn ith[l, i] :
        if i<=1 -> l
        else if l=Nil -> Nil
        else ith[rest[l], i-1].


!*****************************************
!       Utilities --                     !
!  This file contains a set of utility   !
!  rules that are automatically loaded   !
!  at system boot.                       !
!*****************************************



!  System Help function
!-------------------------------------------------------
define{root, "help", newrel{}}.
define{root, "HelpLib", newrel{}}.

HelpLib("?",        "/work/mcarthur/common/summary.help").
```

```
HelpLib("relations","/work/mcarthur/common/relation.help").
HelpLib("functions","/work/mcarthur/common/function.help").
HelpLib("procedure","/work/mcarthur/common/procedure.help").
HelpLib("bugs",     "/work/mcarthur/common/bugs.help").
HelpLib("sample",   "/work/mcarthur/common/sample.help
                     /work/mcarthur/common/*.rul").
HelpLib( "features","/work/mcarthur/common/features.help").


define{root, "HelpRules",
<<

!       this rule may be invoked by an assertion or a call
!       so both cases are covered.

if *help(a:L), (¬IsRel[a] | length[L]= 1) ->
                displayn{"Usage : help{""topic""}"},
                help{"?"}

else if *help(a, x), HelpLib(x, y) ->
        system("more " + y),
        a("OK")

else if *help(a, x) ->
        displayn{"Topic not found."},
        help(a, "?");

>> }.

! Activate the rules
activate{ HelpRules }.

displayn{ "For help, enter help{""?""}." };
displayn{}.


!    bug rules --- add to the bug documentation
!-------------------------------------------------

define {root, "BugRules",
```

```
<<
if *Bugs(a) ->
        displayn{"Describe the bug. End with Cntrl-D :"},
        system{"mail -s 'Bug Report' mcarthur"},
        a("OK")
>>}.

define{root, "Bugs", newrel{}}.

activate{BugRules}.

displayn{"To report a bug, enter Bugs{}."}.



!       CopyRel -- Copy from one relation to another
!       Usage is CopyRel{R1, R2}.
!-------------------------------------------------------

define{root, "CopyRel", newrel{}}.

define{root, "CopyRelRules",
<<

if CopyRel(a, r1, r2), r1(x:y), ¬r2(x:y) ->
        r2(x:y)

else if *CopyRel(a, r1, r2) ->
        a("OK");

>>}.

activate{CopyRelRules}.

!       pp - A pretty printer for objects.
!
!      These rules are required because the standard pretty
!      printer - display - doesn't do lookups on objects.
!
!      define the relations....
```

```
        define{root, "Pp", newrel{}}.
        define{root, "PpAux", newrel{}}.
        define{root, "PpRel", newrel{}}.
        define{root, "PpObj", newrel{}}.
        define{root, "DumpDisplay", newrel{}}.


!       the rules ...

        define{root, "PpRules",
<<

        ! single member is a gocf by the user
        if *Pp(a:L), ¬IsRel[a] ->
                displayn{"Usage : Pp{x1, x2, ...}"};

        if *Pp(a:Nil) ->
                a("OK")

        else if *Pp(a:[x:L]) ->
                PpAux{x},
                Pp(a:L);

        !  Is it a relation ?
        if *PpAux(a, x), IsRel[x] ->
                PpRel{x, newrel{}},
                a("")

        !  or other type of object ?
        else if *PpAux(a, x), IsObj[x] ->
                PpObj(a, x, objval[x]),
                a("")

        !  otherwise just display it
        else if *PpAux(a, x) ->
                displayn{x},
                a("");

        if *PpRel(a, x, temp) ->
```

```
                CopyRel{x, temp},
                DumpDisplay(a, x, temp);

        if *DumpDisplay(a, name, R), *R(x:y) ->
                display{name},
                display{"("},
                display{x:y},
                displayn{")."},
                DumpDisplay(a, name, R)

        else if *DumpDisplay(a, name, R) ->
                a("");

        !       Pretty print objects ...
        !       Functions require a bit of set up

        if *PpObj(a, name, def), Tag[def]="FN" ->
                display{"fn "},
                display{name},
                displayn{def},
                a("")

        !       otherwise, just display def (might be Nil)

        else if *PpObj(a, name, def) ->
                displayn{def};


>> }.

activate{PpRules}.


!       assert list procedures
!       allows one to assert multiple tuples to a relation
!       Usage is Assert{relname, [tuple], [tuple], ...}

define{root, "AssertList", newrel{}}.
define{root, "AssertAux", newrel{}}.
```

```
define{root, "AssertTuple", newrel{}}.

define{root, "AssertListRules",
<<


if *AssertList(a:[r:L]), IsRel[a], IsRel[r] ->
        AssertAux(a, r, L)

else if *AssertList(a:L) ->
        displayn{"Usage : Assert{Relname, ListOfTuples}"},
        a(Nil);

if *AssertAux(a, r, Nil) ->
        a("OK")

else if *AssertAux(a, r, [x:l]) ->
        AssertTuple{r, x},
        AssertAux(a, r, l);

if *AssertTuple(a, r, x), ¬IsList[x] ->
        displayn{"Error in Assert -- tuple not a list", x}

else if *AssertTuple(a, r,  Nil) ->
        a(Nil)

else if *AssertTuple(a, r, [x:l]) ->
        r(x:l),
        a(r);

>>}.

activate{AssertListRules}.
```

APPENDIX D

COMPARATIVE APPLICATIONS: OMEGA, LISP, AND PROLOG

```
**************************************************
*                                                *
*            Pattern Matching Tests              *
*                                                *
*                                                *
**************************************************


!
!       pattern matching benchmark -- Omega
!-----------------------------------------------
define{root, "R1", newrel{}}.
define{root, "R2", newrel{}}.
define{root, "genDat", newrel{}}.
define{root, "mTest",  newrel{}}.

define{root, "MatchRules",
<<

! the data generating rules

if *genDat(a, n1, n2), n1 > n2  ->
        a("OK")
else if *genDat(a, n1, n2) -> {
         R1(n1);
         R2(n1 + n2);
         genDat(a, n1+1, n2);
         };

if *mTest(a) -> {
        R1(9999); R2(9999);
        genDat{1, 1000};
```

```
        system{"date"};
        if R1(x), R2(x) -> {
                system("date");
                displayn{x};
                };
        a("OK");
        }
>>}.

! activate the rules
activate{MatchRules}.


/*
PROLOG pattern matching benchmark
-----------------------------------*/

r1(9999).
r2(9999).

genDat(Lo, Hi) :-
        Lo < Hi,
        I2 is Lo + Hi,
        asserta(r1(Lo)),
        asserta(r2(I2)),
        Lonext is Lo + 1,
        genDat(Lonext, Hi).
genDat(Hi, Hi).

mTest(X) :-
        genDat(1, 1000),
        system("date"),
        r1(X), r2(X),
        system("date").


;
;       pattern matching benchmark -- Franz LISP
```

```
;-----------------------------------------------

(defun genDat (n1 n2)
        (prog (i1)
          (setq i1 n1)
          LOOP
          (cond ((greaterp i1 n2)
                 (return))
                (t (setq r1 (cons i1 r1))
                   (setq r2 (cons (plus i1 n2) r2))
                   (setq i1 (add1 i1))
                   (go LOOP)))))

(defun match ()
        (prog (t1 t2)
          (setq t1 r1)
          LOOP1
          (setq t2 r2)
          LOOP2
          (cond ((null t2)
                 (cond ((null t1) (return nil))
                       (t (setq t1 (cdr t1))
                          (go LOOP1))))
                ((equal (car t1) (car t2))
                 (return (car t1)))
                (t (setq t2 (cdr t2))
                   (go LOOP2)))))

(defun mTest ()
        (setq r1 (list 9999))
        (setq r2 (list 9999))
        (genDat 1 1000)
        (exec date)
        (match)
        (exec date))
```

```
**************************************************
*                                                *
*                  Factorial                     *
*                                                *
*                                                *
**************************************************


!
!       factorial benchmark -- Cmega
!-----------------------------------------------


fn fact[n] :
        if n <= 0 -> 1
        else n * fact[n-1].

!  driver rules

define{root, "Driver", newrel{}}.
define{root, "factTest", newrel{}}.
define{root, "DriverRules",
<<      if *Driver(a, n) -> {
                if n <= 0 -> a("OK")
                else -> {
                        fact[15];
                        Driver(a, n-1)
                        }
                };

!  the CSH date function provides a timer
!  times given to the nearest second

        if *factTest(a, n) -> {
                display{"Omega factorial: "};
                display{n};
                displayn{" calls"};
```

```
                system("date");
                Driver{n};
                system("date");
                };
>>}.

! activate the rules
activate{DriverRules}.



/*
factorial benchmark -- Prolog
-----------------------------------------*/

fact(N, 1) :- N =< 0.
fact(N, F) :-
        N >= 0, fact(N-1, M), F is M * N.

driver(0).
driver(N) :-
        N > 0, fact(15, X), M is N-1, driver(M).

factTest(N) :-
        system("date"),
        driver(N),
        system("date").



;
;  Factorial benchmark -- Franz LISP
;-----------------------------------------

(defun fact (n)
  (cond  ((lessp n 0) 1)
         ((zerop n)  1)
         (t (* n (fact (- n 1))))))
```

```
;  the benchmark driver --
;  (fact 15) executed n times
(defun driver (n)
  (cond ((equal n 0) "OK")
        (t (fact 15)
           (driver (- n 1)))))

;  the CSH date function provides a crude
;  timer to the nearest second.

(defun factTest (n)
  (exec date)
  (driver n)
  (exec date))
```

```
**************************************************
*                                                *
*            Prime Number Sieve                  *
*                                                *
*                                                *
**************************************************
```

```
!
!       sieve benchmark -- Omega
!-------------------------------------------------

define{root, "Primes", newrel{}}.
define{root, "PrimesAux", newrel{}}.
define{root, "sTest", newrel{}}.

fn PurgeMults[x, l] :
        if l = Nil -> Nil
        else if first[l] % x = 0 ->
                PurgeMults[x, rest[l]]
```

```
        else [first[1]:PurgeMults[x, rest[1]]].

define{root, "SieveRules",
<<
if *Primes(a, n) ->
        PrimesAux(a, iota[2, n], Nil);

if *PrimesAux(a, Nil, l) ->
        a(reverse[l])
else if *PrimesAux(a,[x:l1], l2) ->
        PrimesAux(a, PurgeMults[x, l1], [x:l2]);

if *sTest(a) -> {
        system{"date"};
        Primes{350};
        system{"date"};
        a("OK");
        };
>>}.

activate{SieveRules}.


/*
Sieve benchmark -- Prolog
-----------------------------------------------*/

primes(Limit, Ps) :-
         iota(2, Limit, Is),
         sift(Is, Ps).

iota(Lo, Hi, [Lo|Rest]) :-
        Lo =< Hi, M is Lo+1, iota(M, Hi, Rest).
iota(_, _, []).

sift([], []).
sift([I|Is], [I|Ps]) :-
        remove(I, Is, New), sift(New, Ps).
```

```
remove(P, [ ], [ ]).
remove(P, [I|Is], [I|Nis]) :-
        not(0 is I mod P),
        remove(P, Is, Nis).
remove(P, [I|Is], Nis) :-
        0 is I mod P,  remove(P, Is, Nis).

sTest :-
        system("date"),
        primes(350, Ps),
        system("date"),
        print(Ps).
```

```
;
;       sieve benchmark -- Franz LISP
;-----------------------------------------------

(defun iota (n1 n2)
  (cond ((greaterp n1 n2) nil)
        (t (cons n1 (iota (add1 n1) n2)))))

(defun purgeMults (x l)
  (cond ((null l) nil)
        ((zerop (mod (car l) x)) (purgeMults x (cdr l)))
        (t (cons (car l) (purgeMults x (cdr l))))))


(defun primes (n)
        (primesAux (iota 2 n)))

(defun primesAux (l)
        (cond ((null l) nil)
              (t (cons (car l)
                       (primesAux
                           (purgeMults
                               (car l)
                               (cdr l)))))))
```

```
(defun sTest ()
  (exec date)
  (primes 350)
  (exec date))
```

```
**************************************************
*                                                *
*               Quicksort                        *
*                                                *
*                                                *
**************************************************
```

```
!
!       Quicksort benchmark -- Cmega
!-----------------------------------

fn iotaR[n1, n2] :
        if n1 > n2 -> Nil
        else [n2:iotaR[n1, n2-1]].

define{root, "Qsort", newrel{}}.
define{root, "QsortAux", newrel{}}.
define{root, "qTest", newrel{}}.

define{root, "QsortRules",

<<
!       Quick Sort

if *Qsort(a, Nil) ->
        a(Nil);
if *Qsort(a, [x:L]) ->
        QsortAux(a, x, L, Nil, Nil);
```

```
if *QsortAux(a, x, Nil, L1, L2) ->
        a(append[Qsort{L1}, [x:Qsort{L2}]]);
if *QsortAux(a, x, [y:L1], L2, I3) -> {
        if  y <= x ->
                QsortAux(a, x, I1, [y:L2], L3)
        else
                QsortAux(a, x, I1, L2, [y:L3])
        };

if *qTest(a, n) -> {
        system{"date"};
        Qsort{iotaR[1, n]};
        system{"date"};
        a("OK");
        };
>>}.

activate{QsortRules}.


/*      Quicksort Benchmark -- Prolog
----------------------------------------*/

qsort([H|T],S) :-
        split(H,T,A,B),
        qsort(A,A1),
        qsort(B,B1),
        append(A1,[H|B1],S).
qsort([],[]).

split(H,[A|X],[A|Y],Z) :-
        A < H,
        split(H,X,Y,Z).
split(H,[A|X],Y,[A|Z]) :-
        H < A,
        split(H,X,Y,Z).
split(_,[],[],[]).
```

```
append([ ],L,L).
append([H|T],L,[H|V]) :-
        append(T,L,V).


iotaR(Lo, Hi, [Hi|Rest]) :-
        Lo =< Hi, M is Hi-1, iotaR(Lo, M, Rest).
iotaR(_, _, [ ]).


qTest(N) :-
        system("date"),
        iotaR(1, N, L),
        qsort(L, S),
        system("date"),
        print(S).


;
;  Quicksort Benchmark -- Franz LISP
;-----------------------------------------------

(defun qsort(l)
   (cond ((null l) nil)
         (t (setq s (split (car l) (cdr l)))
            (append (qsort (car s))
                    (cons (car l) (qsort (cadr s)))))))

(defun split (x l)
  (prog (s)
    (cond ((null l) (return (list nil nil))))
    (setq s (split x (cdr l)))
    (cond ((lessp (car l) x)
           (return (list (cons (car l) (car s)) (cadr s))))
          (t
           (return (list (car s)
```

```
                    (cons (car l) (cadr s))))))))
```

```
(defun iotaR (n1 n2)
  (cond ((greaterp n1 n2) nil)
        (t (cons n2 (iotaR n1 (sub1 n2))))))

(defun qTest (n)
  (exec date)
  (qsort (iotaR 1 n))
  (exec date))
```

```
*****************************************************
*                                                   *
*          Missionaries and Cannibals:              *
*          A State Search Problem                   *
*                                                   *
*                                                   *
*****************************************************
```

```
!
!  Missionaries and Cannibals -- Omega
!  Generate and test search strategy
!-----------------------------------------------

define{root, "mc", newrel{}}.
define{root, "misCan", newrel{}}.
define{root, "PpL", newrel{}}.

define{ root, "misCanRules",
        <<
        if *mc(nM, nC) ->
                misCan(1,nM,nC,nM,nC,0,0,Nil);
```

```
if misCan(s,nM,nC,0,0,nM,nC,path)
->      {
        purge{misCan};
        PpL{[[s,0,0,nM,nC]:path]};
        displayn{"Finis"}
        }
else if  *misCan(s,nM,nC,mL,cL,mR,cR,path),
        (mL >= 0  &  mL <= nM &
         cL >= 0  &  cL <= nC &
         mR >= 0  &  mR <= nM &
         cR >= 0  &  cR <= nC &
         (mL >= cL | mL = 0)  &
         (mR >= cR | mR = 0)),
         ¬member[[s,mL,cL,mR,cR],path]
->
             misCan((0-s),nM,nC,mL-s,cL,mR+s,cR,
                       s,mL,cL,mR,cR :path ),]
             misCan((0-s),nM,nC,mL,cL-s,mR,cR+s,
                       s,mL,cL,mR,cR :path ),]
             misCan((0-s),nM,nC,mL,cL-2*s,mR,cR+2*s,
                       s,mL,cL,mR,cR :path ),]
             misCan((0-s),nM,nC,mL-2*s,cL,mR+2*s,cR,
                       s,mL,cL,mR,cR :path ),]
             misCan((0-s),nM,nC,mL-s,cL-s,mR+s,cR+s,
                       s,mL,cL,mR,cR :path )  ]

else if *misCan(s,nM,nC,mL,cL,mR,cR,path) ->;

!  a small pretty printer for the output list

if *PpL(a, Nil) ->
else if *PpL(a, [x:L]) -> {
        PpL{L};
        displayn{x};
        }

>> }.
```

```
activate{ misCanRules }.


/*
Missionaries and Cannibals -- Prolog
-----------------------------------------------*/

mc(Nm,Nc) :- misCan(-1,Nm,Nc,0,0,Nm,Nc,[ ]).

misCan(S,Nm,Nc,Nm,Nc,0,0,Path) :-
        Nm >= Nc, ppl([[S,Nm,Nc,0,0]|Path]).
misCan(S,Nm,Nc,Ml,Cl,Mr,Cr,Path) :-
        safe(Nm,Nc,Ml,Cl,Mr,Cr),
        equal([S,Ml,Cl,Mr,Cr],P),
        not(member(P,Path)),
        S1 is -S,
        possible(S,Ml,Cl,Mr,Cr,Ml1,Cl1,Mr1,Cr1),
        misCan(S1,Nm,Nc,Ml1,Cl1,Mr1,Cr1,[P|Path]).

safe(Nm,Nc,Ml,Cl,Mr,Cr) :-
        Ml >= 0, Ml =< Nm,
        Cl >= 0, Cl =< Nc,
        (Ml >= Cl ; Ml =:= 0),
        (Mr >= Cr ; Mr =:= 0).

possible(S,Ml,Cl,Mr,Cr,Ml1,Cl1,Mr1,Cr1) :-
        (Ml1 is Ml-2*S,
         Mr1 is Mr+2*S,
         Cl1 is Cl,
         Cr1 is Cr);

        (Ml1 is Ml,
         Mr1 is Mr,
         Cl1 is Cl-2*S,
         Cr1 is Cr+2*S);
```

```
        (Ml1 is Ml-S,
         Mr1 is Mr+S,
         Cl1 is Cl-S,
         Cr1 is Cr+S);

        (Ml1 is Ml-S,
         Mr1 is Mr+S,
         Cl1 is Cl,
         Cr1 is Cr);

        (Ml1 is Ml,
         Mr1 is Mr,
         Cl1 is Cl-S,
         Cr1 is Cr+S).


member(X,[ ]) :- !,fail.
member(X,[X|L]).
member(X,[Y|L]) :- member(X,L).

equal(X,X).

ppL([ ]).
ppL([X|L]) :- print(X), nl, ppL(L).
```

APPENDIX E

OMEGA APPLICATION EXAMPLES

```
!************************************************************!
!                                                            !
!       Monte Carlo Simulation for 3 node message network.   !
!                                                            !
!************************************************************!

!       the relations

define{root, "Begin", newrel{}}.
define{root, "End", newrel{}}.
define{root, "Clock", newrel{}}.
define{root, "NextTime", newrel{}}.
define{root, "LastTime", newrel{}}.
define{root, "Event", newrel{}}.
define{root, "ProcessEvents", newrel{}}.
define{root, "ProcessEventsAux", newrel{}}.
define{root, "Summary", newrel{}}.
define{root, "Status",   newrel{}}.
define{root, "BusyTime", newrel{}}.
define{root, "QLength", newrel{}}.
define{root, "Stats", newrel{}}.
define{root, "BusyStats", newrel{}}.
define{root, "QStats", newrel{}}.
define{root, "Totals", newrel{}}.
define{root, "NodeTotals", newrel{}}.
define{root, "JobTotals", newrel{}}.
define{root, "JobTotalsAux", newrel{}}.
define{root, "DisplayLine", newrel{}}.
define{root, "Start",  newrel{}}.
define{root, "Arrive", newrel{}}.
define{root, "NewArrival", newrel{}}.
```

```
define{root, "Service", newrel{}}.
define{root, "StartTime", newrel{}}.
define{root, "Depart", newrel{}}.
define{root, "ExitTime", newrel{}}.
define{root, "InService", newrel{}}.
define{root, "Q", newrel{}}.
define{root, "Eng", newrel{}}.
define{root, "CalcNextArr", newrel{}}.
define{root, "CalcDeparture", newrel{}}.
define{root, "CalcRoute", newrel{}}.
define{root, "NewJob", newrel{}}.
define{root, "JobServiceT", newrel{}}.
define{root, "JobCount", newrel{}}.
define{root, "GenArrT", newrel{}}.
define{root, "GenServT", newrel{}}.
define{root, "GenRoute", newrel{}}.
define{root, "ServiceTime", newrel{}}.
define{root, "ArrInterval", newrel{}}.
define{root, "PossibleRoutes", newrel{}}.
define{root, "Seed", newrel{}}.
define{root, "Rnd", newrel{}}.
define{root, "RndList", newrel{}}.
define{root, "WorkList", newrel{}}.

!       the Nodes

define{root, "Node1", newobj{}}.
define{root, "Node2", newobj{}}.
define{root, "Node3", newobj{}}.
define{root, "Out",   newobj{}}.


!       initalize MonteCarlo tables

!       Service times (message transmission times)

ServiceTime(1,  0, 12).
```

```
ServiceTime(2, 13, 32).
ServiceTime(3, 33, 79).
ServiceTime(4, 80, 92).
ServiceTime(5, 93, 99).

!       Interarrival times

ArrInterval(Node1, 1,  0, 16).
ArrInterval(Node1, 2, 17, 24).
ArrInterval(Node1, 3, 25, 49).
ArrInterval(Node1, 4, 50, 91).
ArrInterval(Node1, 5, 92, 99).

ArrInterval(Node2, 1,  0, 28).
ArrInterval(Node2, 2, 29, 70).
ArrInterval(Node2, 3, 71, 85).
ArrInterval(Node2, 4, 86, 92).
ArrInterval(Node2, 5, 93, 99).


!       Routing table

PossibleRoutes(Node1, Node2,  0, 29).
PossibleRoutes(Node1, Node3, 30, 59).
PossibleRoutes(Node1, Out,   60, 99).

PossibleRoutes(Node2, Node1,  0, 29).
PossibleRoutes(Node2, Node3, 30, 49).
PossibleRoutes(Node2, Out,   50, 99).

PossibleRoutes(Node3, Out,    0, 99).


!       Random number table

RndList([

97,95,12,11,90,49,57,13,86,81,
02,92,75,91,24,58,39,22,13,02,
80,67,14,99,16,89,96,63,00,04,
```

96,76,20,28,72,12,77,23,79,46,
55,64,82,61,73,94,26,18,37,31,
50,02,74,70,16,85,95,32,85,67,
29,53,08,33,81,34,30,21,24,25,
58,16,01,91,70,07,50,13,18,24,
51,16,69,67,16,53,11,06,36,10,
04,55,36,97,30,99,80,10,52,40,
86,54,35,61,59,89,64,97,16,02,
24,23,52,11,59,10,88,68,17,39,
39,36,99,50,74,27,69,48,32,68,
60,71,41,25,90,93,07,24,29,59,
65,88,48,06,68,92,70,97,02,66,
44,74,11,60,14,57,08,54,12,90,
93,10,95;80,32,50,40,44,08,12,
20,46,36,19,47,78,16,90,59,64,
86,54,24,88,94,14,58,49,80,79,
12,88,12,25,19,70,40,06,40,31,
42,00,50,24,60,90,69,60,07,86,
29,98,81,68,61,24,90,92,32,68,
36,63,02,37,89,40,81,77,74,82,
01,77,82,78,20,72,35,38,56,89,
41,69,43,37,41,21,36,39,57,80,
54,40,76,04,05,01,45,84,55,11,
68,03,82,32,22,80,92,47,77,62,
21,31,77,75,43,13,83,43,70,16,
53,64,54,21,04,23,85,44,81,36,
91,66,21,47,95,69,58,91,47,59,
48,72,74,40,97,92,05,01,61,18,
36,21,47,71,84,46,09,85,32,82,
55,95,24,85,84,51,61,60,62,13,
70,27,01,88,84,85,77,94,67,35,
38,13,66,15,38,54,43,64,25,43,
36,80,25,24,92,98,35,12,17,62,
98,10,91,61,04,90,05,22,75,20,
50,54,29,19,26,26,87,94,27,73

```
]).

WorkList(Nil).

!       Intialize Queues

Q(Node1, Nil).
Q(Node2, Nil).
Q(Node3, Nil).

!       the Precedence function
!       Scheme is (for events @time=t, node=n)
!               Departures have top priority
!               Between occurrences of same type of event
!               JobN1 precedes JobN2 if N1 < N2.
!       This function replaces a sort on the event list.

fn Precedence[e1, j1, e2, j2] :
        if e1=e2 & j1<j2 -> "TRUE"
        else if e1=Depart & e2 ¬= Depart -> "TRUE"
        else Nil.

!       The Rules

define{root, "SimRules",
<<

!
!     Begin -- Begins the simulation and sets up relations
!

if *Begin(a, n) -> {
        JobCount(0, 0, n);
        LastTime(0);
```

```
        InService(Node1, "--");
        InService(Node2, "--");
        InService(Node3, "--");

        BusyTime(Node1, 0);
        BusyTime(Node2, 0);
        BusyTime(Node3, 0);

        QLength (Node1, 0);
        QLength (Node2, 0);
        QLength (Node3, 0);

        Event(0, Start, "--", "--");

        Clock(0);
        a("OK");
        };

!
!       End -- Cleanup for further runs
!

if *End(a) -> {
        purge{JobCount};
        purge{InService};
        a(End);
        };

!
!       the clock rules -- drives the simulation
!

if *Clock(999), *LastTime(t) -> {
        Totals{t};
        End{};
        }


else if *Clock(t2), *LastTime(t1) -> {
```

```
        Stats{t1, t2};
        displayn{
"----------------------------------------"};
        display{"Time : "};
        displayn{t2};
        displayn{"Events :"};
        ProcessEvents{t2};
        Summary{};
        LastTime(t2);
        Clock(NextTime{999});       ! restart the clock
        };

if *NextTime(a, t1), Event(t2, e, n, j), t2 < t1 ->
        NextTime(a, t2)

else if *NextTime(a, t) ->
        a(t);

!
!       Process all events for time t
!

if *ProcessEvents(a, t), Event(t, e1, n1, j1) -> {
        ProcessEventsAux{t, e1, n1, j1};
        ProcessEvents(a, t);
        }

else if *ProcessEvents(a, t) ->
        a(t);

if *ProcessEventsAux(a, t, e1, n1, j1),
   Event(t, e2, n2, j2),
   Precedence[e2, j2, e1, j1] ->
        ProcessEventsAux(a, t, e2, n2, j2)

else if *ProcessEventsAux(a, t, e, node, job) -> {
        ¬Event(t, e, node, job);
        e{t, node, job};
```

```
        display{"       "};
        displayn{e, node, job};
        a(t);
        };


!
!       Summarize the current status for all nodes
!
if *Summary(a) -> {
        Status{Node1}, Status{Node2}, Status{Node3};
        displayn{};
        displayn{"Pending Events :"};
        Pp{Event};
        displayn{};
        a("");
        };



if *Status(a, node), InService(node, job), Q(node, l)  ->
        {
        displayn{node};
        display{"       Job in service : "};
        displayn{job};
        display{"       Jobs in queue  : "};
        if l=Nil -> displayn{"--"}
         else displayn{l};
        a(node);
        };

!
!       Keep running totals for stats on each node.
!       The relations are :
!               BusyTime(node, t) -- Cumulative idle time
!               QLength(node,  n) -- Time weighted Q length
```

```
!
!
if *Stats(a, t1, t2) -> {
        BusyStats{t1, t2, Node1};
        BusyStats{t1, t2, Node2};
        BusyStats{t1, t2, Node3};
        QStats{t1, t2, Node1};
        QStats{t1, t2, Node2};
        QStats{t1, t2, Node3};
        a(t1);
        };


if *BusyStats(a, t1, t2, node), InService(node, "--") ->
        a(node)

else if *BusyStats(a, t1, t2, ncde), *BusyTime(node, t) ->
        BusyTime(node, t+t2-t1),
        a(node);


if *QStats(a, t1, t2, node), *QIength(node,n), Q(node,1) ->
        QLength(node, n + (t2-t1)*length[1]),
        a(node);

!
!       Totals -- Display stat summary at end of simulation
!

if *Totals(a, t) -> {
        display{"       Total time : "};
        displayn{t};
        displayn{};
        DisplayLine{["Node", "Busy", "QLen"]};
        DisplayLine{["--------------------"]};
        NodeTotals{Node1};
        NodeTotals{Ncde2};
```

```
        NodeTotals{Node3};
        displayn{};
        DisplayLine{["Job", "Time"]};
        DisplayLine{["----------"]};
        JobTotals{Nil, 0};
        a(t);
        };

if *NodeTotals(a, node), *BusyTime(node, t),
   *QLength(node, n) -> {
        DisplayLine{[node, t, n]};
        a(node);
        };

if *JobTotals(a, l, n),
   *StartTime(j, t1), *ExitTime(j, t2) ->
        JobTotals(a, cons[[j, t2-t1], l], n+t2-t1)

else if *JobTotals(a, l, n) -> {
        JobTotalsAux{l};
        displayn{};
        DisplayLine{["Total", n]};
        a(JobTotals);
        };
if *JobTotalsAux(a, Nil) ->
        a(JobTotalsAux)

else if *JobTotalsAux(a, l) -> {
        DisplayLine{first[l]};
        JobTotalsAux(a, rest[l]);
        };



if *DisplayLine(a, Nil) -> {
        displayn{};
        a(DisplayLine);
```

```
        }
else if *DisplayLine(a, l) -> {
        display{"       "};
        display{first[l]};
        DisplayLine(a, rest[l]);
        };
!
!       the events -- Start, Arrive, Depart
!
if *Start(a, t, x, y) -> {
        CalcNextArr{t, Node1};
        CalcNextArr{t, Node2};
        a(t);
        };
if *Arrive(a, t, node, "NewJob") -> {
        NewArrival{t, node, NewJob{}};
        a(t)
        }
else if *Arrive(a, t, Out, job) -> {
        ExitTime(job, t);
        a(t)
        }
else if *Arrive(a, t, node, job) -> {
        Service{t, node, job};
        a(t);
        };
if *NewArrival(a, t, node, job) -> {
        StartTime(job, t);
        CalcNextArr{t, node};
        JobServiceT(job, GenServT{Rnd{}});
        Service{t, node, job};
```

```
        a(t);
        };

if *Service(a, t, node, job), *InService(node, "--") -> {
        InService(node, job);
        CalcDeparture{t, node, job};
        a(t)
        }

else if *Service(a, t, node, job) -> {
        Enq{node, job};
        a(t);
        };

if *Depart(a, t, node, job),
    Q(node, Nil), *InService(node, job) ->
        InService(node, "--"),
        a(t)

else if *Depart(a, t, node, job),
        *Q(node, l), *InService(node, job) ->
        {
        Q(node, rest[l]);
        InService(node, "--");
        Service{t, node, first[l]};
        a(t);
        };



if *CalcNextArr(a,t,ncde), JobCcunt(n1,n2,max), n1>=max ->
        a(t)

else if *CalcNextArr(a,t,node), *JobCount(n1,n2,max) ->
        {
        JobCount(n1+1, n2, max);
        Event(t+GenArrT{node,Rnd{}},Arrive,node,"NewJob");
        a(t);
```

```
        };

if *CalcDeparture(a,t,node,job), JobServiceT(job,servT) ->
        {
        Event(t+servT,Depart,node,job);
        Event(t+servT,Arrive,GenRoute{node,Rnd{}},job);
        a(t);
        };


if *Enq(a, node, job), *Q(node, l) ->
        Q(node, append[l, [job]]),
        a(node);

if *NewJob(a), *JobCount(n1, n2, max) ->
        JobCount(n1, n2+1, max),
        a("Job"+int_str[n2+1]);



!
!       parameter generators
!       GenServT -- Generate a service time for a message
!       GenArrT  -- Generate an interarrival time for a job
!       GenRoute -- Generate the next node for a departure
!

if *GenServT(a, x), ServiceTime(t, i1, i2),
   (x >= i1) & (x <= i2) ->
        a(t);

if *GenArrT(a, node, x), ArrInterval(node, t, i1, i2),
   (x >= i1) & (x <= i2) ->
        a(t);
```

```
if *GenRoute(a, node, x), PossibleRoutes(node, n2, i1, i2),
   (x >= i1) & (x <= i2) ->
        a(n2);

!
!       The random number generator ....
!       The Rnd procedure takes the first number from the
!       list of random numbers in WorkList.
!       Seed makes the WorkList the portion of the
!       RndList with the ith member as the first member.
!       If the list of random numbers is exhausted,
!       the WorkList is set back to the beginning of
!       the original list of random numbers.
!

if *Seed(a, n), RndList(l), *WorkList(wl) ->
        WorkList(ith[l, n]),
        a(n);

if *Rnd(a), *WorkList(Nil), RndList(l) ->
        WorkList(rest[l]),
        a(first[l])

else if *Rnd(a), *WorkList(l) ->
        WorkList(rest[l]),
        a(first[l]);

>>}.

!       Activate the rules...

activate{SimRules}.
```

```
!*********************************************
!                                             !
!              Towers of Hanoi                !
!                                             !
!*********************************************

!       define the relations
define{root, "Hanoi", newrel{}}.
define{root, "HanoiAux", newrel{}}.

!       The rules
define{root, "HanoiRules",
<<

if *Hanoi(a, n) ->
        HanoiAux(a, n, "A", "C", "B");

if *HanoiAux(a, 1, from, to, aux) -> {
        display{"Move disk 1 from peg "};
        display{from};
        display{" to peg "};
        displayn{to};
        a("")
        }

else if *HanoiAux(a, n, from, tc, aux) -> {
        HanoiAux{n-1, from, aux, to};
        display{"Move disk "};
        display{n};
        display{" from peg "};
        display{from};
        display{" to peg "};
        displayn{to};
        HanoiAux{n-1, aux, to, from};
        a("");
        }
>> }.
```

```
activate{HanoiRules}.

displayn{"Towers of Hanoi : Usage : Hanoi{n}"}.


!*********************************************************
!               The Sieve of Eratosthenes                 !
!                                                         !
!For a description of this algorithm, see                 !
!'Eratosthenes revisited : Once More through the Sieve',!
!by  Jim and Gary Gilbreath, Byte, Jan 83, p 283.         !
!                                                         !
!*********************************************************!


define{root, "Sieve", newrel{}}.
define{root, "SieveAux", newrel{}}.
define{root, "Iota", newrel{}}.
define{root, "PurgeMultiples", newrel{}}.

!       the rules

define{root, "SieveRules",
<<

if *Sieve(a, n) -> system("date"),
        SieveAux(a, 0, n-1, Iota{0, n-1, newrel{}});

if *Iota(a, n1, n2, r), n1 > n2 ->
        a(r)

else if *Iota(a, n1, n2, r) ->
        r(n2),
        Iota(a, n1, n2-1, r);

if *SieveAux(a, n1, n2, r), n1 > n2  ->
        a("OK")

else if *SieveAux(a, n1, n2, r), r(n1) -> {
```

```
        display{"Prime : "};
        displayn{2*n1 + 3};
        PurgeMultiples{2*n1 + 3, 3*n1 + 3, n2, r};
        SieveAux(a, n1+1, n2, r);
        }
else if *SieveAux(a, n1, n2, r) ->
        SieveAux(a, n1+1, n2, r);
if *PurgeMultiples(a, prime, sum, n, r), sum > n ->
        a(r)
else if *PurgeMultiples(a, prime, sum, n, r), *r(sum) ->
        PurgeMultiples(a, prime, prime+sum, n, r)
else if *PurgeMultiples(a, prime, sum, n, r) ->
        PurgeMultiples(a, prime, prime+sum, n, r);
>> }.
activate{SieveRules}.
displayn{"The sieve is loaded : Usage : Sieve{n}"}.

!*************************************************************
!                                                            !
!       Rules and asscciated definitions for a universal     !
!       interpreter/pretty printer for a small language of!
!       arithmetic expressions.                              !
!                                                            !
!************************************************************!

!       definitions
define{root, "Small", newrel{}};
define{root, "Wait",  newrel{}};
define{root, "Eval", newrel{}};
define{root, "Value", newrel{}};
```

```
define{root, "Appl", newrel{}};
define{root, "Op", newrel{}};
define{root, "Left", newrel{}};
define{root, "Right", newrel{}};
define{root, "Con", newrel{}};
define{root, "Litval", newrel{}};
define{root, "Meaning", newrel{}};
define{root, "Template", newrel{}};

!       scme objects
define{root, "N1", newobj{}};
define{root, "N2", newobj{}};
define{root, "N3", newobj{}};
define{root, "N4", newobj{}};
define{root, "N5", newobj{}}.


!       functions
fn Id[x] : x;
fn Sum[x,y] : x + y;
fn Product[x,y] : x * y;
fn upSum[x,y] : "(" + x + " + " + y + ")";
fn upProd[x,y] : "(" + x + " x " + y + ")".


!       initialize the database

Appl(N1);
Op("x", N1);
Left(N2, N1);
Right(N3, N1);

Appl(N2);
Op("+", N2);
Left(N4, N2);
Right(N5, N2);

Con(N3);
```

```
Litval(6,N3);
Con(N4);
Litval(3,N4);
Con(N5);
Litval(5,N5);

Meaning(Sum, "+");
Meaning(Product, "x");
Meaning(Id, "lit");

Template(upSum, "+");
Template(upProd, "x");
Template(int_str, "lit");


!       the Rules

define{root, "SmallRules",
<<
        ! Driver rule
        if *Small(node)
                ->
                Eval(Meaning, node),
                Eval(Template, node),
                Wait(node);

        if *Value(Meaning, x, ncde),
           *Value(Template, y, node),
           *Wait(node)
        ->
                displayn(x),
                displayn(y);

        ! Inheritance rules

        ! Leaf node
        if
        *Eval(class, e),
```

```
        Con(e),
        Litval(v,e),
        class(f, "lit")
                ->
                Value(class, f[v], e);

        ! Appl node
        if
        *Eval(class, e),
        Appl(e),
        Left(x,e), Right(y,e)
                ->
                Eval(class, x),
                Eval(class, y);

        ! Synthesis rule

        if
        *Value(class, u,x),
        *Value(class, v,y),
        Appl(e),
        Op(n,e),
        Left(x,e),
        Right(y,e),
        class(f, n)
                ->
                Value(class, f[u,v], e)

>> }.

!       activate the rules

activate{ SmallRules }.
displayn{"Small interpreter loaded : u Small(N1)"}.
```

LIST OF REFERENCES

1. Backus, J., "Can Programming Be Liberated from the von Neumann Style? A Functional Style and Its Algebra of Programs," CACM, Vol. 21, No. 8, pp. 613-641, August 1978.

2. McCarthy, J., "Recursive Functions of Symbolic Expressions and Their Computation by Machine, I," CACM, Vol. 3, No. 4, pp. 185-195, August 1960.

3. Turner, D. A., "Recursion Equations as a Programming Language," in Functional Programming and Its Applications: An Advanced Course (Darlington, J., Henderson, P., and Turner, D. A., eds.), Cambridge, pp. 1-28, 1982.

4. Dahl, O. and Nygaard, K., "SIMULA--An ALGOL-Based Simulation Language," CACM, Vol. 9, No. 9, pp. 671-678, September 1966.

5. Kay, A., "Microelectronics and the Personal Computer," Scientific American, Vol. 237, No. 3, pp. 230-244, September 1977.

6. Post, E., "Formal Reductions of the General Combinatorial Decision Problem," Am Jnl Math, Vol. 65, No. 2, pp. 197-215, April 1943.

7. Newell, A. and Simon, H., Human Problem Solving, Prentice-Hall, 1972.

8. Hewitt, C., "Procedural Embedding of Knowledge in PLANNER," Proc. 2nd IJCAI, London, pp. 167-182, 1971.

9. Lenat, D., "Automated Theory Formation in Mathematics," Proc. 5th IJCAI, Cambridge, Mass., pp. 833-842, 1977.

10. Shortliffe, E. H., Computer-based Medical Consultations: MYCIN, American Elsevier, 1976.

11. Feigenbaum, E. A., Buchanan, B. G., and Lederberg, J., "On Generality and Problem Solving: A Case Study Using the DENDRAL Program," in Machine Intelligence, Vol. 6 (Meltzer, B. and Michie, D., eds.), American Elsevier, pp. 165-190, 1971.

12. Duda, R. O., Hart, P. E., Nilsson, N. J., and Sutherland, G., "Semantic Network Representations in Rule-Based Systems," in Pattern-Directed Inference Systems (Waterman, D. A., and Hayes-Roth, F., eds.), Academic Press, pp. 203-221, 1978.

13. Kowalski, R., Logic for Problem Solving, North-Holland, 1979.

14. Naval Postgraduate School Report NPS 52-83-001, A View of Object-Oriented Programming, by B. J. MacLennan, February 1983.

15. MacLennan, B. J., "Values and Objects in Programming Languages," SIGPLAN Notices, Vol. 17, No. 12, pp. 70-79, December 1982.

16. Codd, E. F., "A Relational Model of Data for Large Shared Data Banks," CACM, Vol. 13, No. 6, June 1970.

17. Dennis, J. B., and Van Horn, E. C., "Programming Semantics for Multiprogrammed Computations," CACM, Vol. 9, No. 3, pp. 143-155, March 1966.

18. Davis, R. and King, J., "An Overview of Production Systems," in Machine Intelligence, Vol. 8 (Elcock, P. W. and Michie, D., eds.), Wiley, pp. 300-332, 1977.

19. MacLennan, B. J., Principles of Programming Languages: Design, Evaluation, and Implementation, Holt, Rinehart, and Winston, 1983.

20. Foderaro, J. K., The Franz LISP Manual, Regents of the University of California, 1980.

21. Clockskin, W. F., and Mellish, C. S., Programming in Prolog, Springer-Verlag, 1981.

22. Kernigan, B. W. and Ritchie, D. M., The C Programming Language, Prentice-Hall, 1978.

23. Lesk, M. E., and Schmidt, E., Lex - A Lexical Analyzer Generator, Bell Laboratories, undated.

24. Johnson, S. C., Yacc: Yet Another Compiler-Compiler, Bell Laboratories, 1978.

25. MacLennan, B. J., Functional Programming Methodology: Theory and Practice, to be published by Addison-Wesley.

26. University of California, The UNIX Programmer's Manual, 4.2 Berkeley Software Distribution, 1983.

27. Barrett, W. A., and Couch, J. D., Compiler Construction: Theory and Practice, Science Research Associates, 1979.

28. Aho, A. V., Hopcroft, J. E., and Ullman, J. D., Data Structures and Algorithms, Addison-Wesley, 1983.

29. Baker, H. G. Jr., "Optimizing the Allocation and Garbage Collection of Spaces," in Artificial Intelligence: An MIT Perspective, Vol. 2, MIT Press, pp. 391-396, 1979.

30. Bobrow, D. G., and Clark, D. W., "Compact Encoding of List Structures," ACM TOPLAS, Vol. 1, No. 2, pp. 266-286, October 1979.

31. SRI International, C-Prolog User's Manual, Version 1.5, F. Pereira, ed., undated.

32. Warren, D. H., Pereira, L. M., and Pereira, F., "Prolog--The Language and Its Implementation Compared with LISP," SIGPLAN Notices, Vol. 12, No. 8, pp. 109-115, August 1977.

33. Rich, E., Artificial Intelligence, McGraw-Hill, 1983.

34. Rosenschein, S. J., "The Production System: Architecture and Abstraction," in Pattern-Directed Inference Systems (Waterman, D. A., and Hayes-Roth, F., eds.), Academic Press, pp. 525-538, 1978.

35. Davis, R., "Meta-Rules: Reasoning About Control", Artificial Intelligence, Vol. 15, No. 3, pp. 179-222, December 1980.

36. Tick, E., and Warren, D. H. D., "Towards a Pipelined Prolog Processor," in IEEE 1984 International Symposium on Logic Programming, IEEE Computer Society Press, pp. 29-40, 1984.

BIBLIOGRAPHY

Foster, J. M., "Programming Language Design For the Representation of Knowledge," in Machine Intelligence, Vol. 8 (Elcock, E. W., and Michie, D., eds.), Wiley, pp. 209-222, 1976.

Goldberg, A., and Robson, D., Smalltalk-80: The Language and Its Implementation, Addison-Wesley, 1983.

Hayes-Roth, F., Waterman, D. A., and Lenat, D. B., "Principles of Pattern-Directed Inference Systems," in Pattern-Directed Inference Systems (Waterman, D. A., Hayes-Roth, F., eds.), Academic Press, pp. 577-601, 1978.

McCarthy, J., and others, LISP 1.5 Programmer's Manual, MIT Press, 1962.

McDermott, J., Newell, A., and Moore, J., "The Efficiency of Certain Production System Implementations," in Pattern-Directed Inference Systems (Waterman, D. A., and Hayes-Roth, F., eds.), Academic Press, pp. 155-176, 1978.

McDermott, J., and Forgy, C., "Production System Conflict Resolution Strategies," in Pattern-Directed Inference Systems (Waterman, D. A., and Hayes-Roth, F., eds.), Academic Press, pp. 177-199, 1978.

Rieger, C., "Spontaneous Computation and Its Role in AI Modeling," in Pattern-Directed Inference Systems (Waterman, D. A., and Hayes-Roth, F., eds.), Academic Press, pp. 69-97, 1978.

Robinson, J. A., "A Machine-Oriented Logic Based on the Resolution Principle," Journal of the ACM, Vol. 12, No. 1, pp. 23-41, January 1965.

Steele, G. L., "Data Representation in PDP-10 MacLISP," Proc. of the MACSYMA User's Conference, NASA, pp. 203-214, 1977.

Warren, D. H. D., "Higher-Order Extensions to Prolog: Are They Needed?," in Machine Intelligence, Vol. 10 (Hayes, J. E., Michie, D., and Pao, Y., eds.), Wiley, pp. 441-454, 1982.

Waterman, D. A., and Hayes-Roth, F., "An Overview of Pattern-Directed Inference Systems," in Pattern-Directed Inference Systems (Waterman, D. A., and Hayes-Roth, F., eds.), Academic Press, pp. 3-22, 1978.

Winston, P. H., and Horn, B. K. P., LISP, Addison-Wesley, 1981.

INITIAL DISTRIBUTION LIST

		No. Copies
1.	Defense Technical Information Center Cameron Station Alexandria, Virginia 22314	2
2.	Library, Code 0142 Naval Postgraduate School Monterey, California 93943	2
3.	Department Chairman, Code 52 Department of Computer Science Naval Postgraduate School Monterey, California 93943	2
4.	Computer Technology Programs Code 37 Naval Postgraduate School Monterey, California 93943	1
5.	Professor Daniel Davis Department of Computer Science, Code 52 Naval Postgraduate School Monterey, California 93943	1
6.	Captain H. M. McArthur, USMC 135 Johnson Street Red Springs, North Carolina 28377	2
7.	Professor J. M. Wozencraft Department of Electrical and Computer Engineering, Code 62 Naval Postgraduate School Monterey, California 93943	1
8.	Dr. Robert Grafton Code 433 Office of Naval Research 800 N. Quincy Arlington, Virginia 22217	1
9.	Mr. Dennis Hall 2 Ivy Drive Orinda, California 94563	1
10.	Dr. David W. Mizell Office of Naval Research 1030 East Green Street Pasadena, California 91106	1
11.	Professor Rudolf Bayer Institut fur Informatik Technische Universitat Postfach 202420 D-8000 Munchen 2 West Germany	1

12. Mr. Ron Laborde 1
INMOS
Whitefriars
Lewins Mead
BRISTOL
United Kingdom

13. Mr. Lynwood Sutton 1
Code 424, Building 600
Naval Ocean Systems Center
San Diego, California 92152

14. Mr. Jeffrey Dean 1
Advanced Information and Decision Systems
201 San Antonio Circle, Suite 286
Mountain View, California 94040

15. Mr. David Lefkovitz 1
310 Cynwyd Rd.
Bala Cynwyd, Pennsylvania 19004

16. Dr. Robert Balzer 1
USC Information Sciences Institute
Suite 1001
4676 Admiralty Way
Marina del Rey, California 90291

17. Mr. Jack Fried 1
Manager, Software Technology
Grumman Aerospace Corporation
Bethpage, New York 11714

18. Mr. Ronald E. Joy 1
Manager, Administration and Technical Services
Honeywell, Inc.
Computer Sciences Center
10701 Lyndale Avenue South
Bloomington, Minnesota 55402

19. LtCol. David Melchar, USMC, Code 0309 1
United States Marine Corps Representative
Naval Postgraduate School
Monterey, California 93943

20. Mr. A. Dain Samples 1
Computer Science Division--EECS
University of California, Berkeley
Berkeley, California 94720

21. Professor S. Ceri 1
Laboratorio di Calcolatori
Departimento di Elettronica
Politecnico di Milano
20133 - Milano
Italy

212672

Thesis
M127
c.1

McArthur
The design and implementation of an object-oriented, production-rule interpreter.